BASH cheat sheet - Level 2

Miscellaneous

\ Escape character. It preserves the literal value of the next character that follows, with the exception of newline.

` ***command***` The backtick (`) is a command substitution.
echo The current working directory is: `pwd`
>The current working directory is: /*home*/*user*/*path*

The text between a pair of backtick is executed by the shell before the main command and is then replaced by the output of that execution. The syntax **$(*command*)** is generally preferable.

$ It introduces parameter expansion, command substitution, or arithmetic expansion. The parameter name or symbol to be expanded may be enclosed in braces.

Using variables

variable*=*value
Assign a value *value* to the variable *variable*. The variable scope is restricted to the shell.

local *variable*=*value*
Assign a value *value* to the local variable *variable*. It doesn't come out a curly bracket area.

export *variable*=*value*
Make the variable *name* available to the shell and sub-processes.

***variable*=$(*command*)**
Assign the output of *command* to *variable*.

${#*variable*}
Length of the value contained by the variable.

${*variable*:*N*}
Keep the character of the value contained by variable after the **N**th.

${*variable*:*N*:*length*}
Substring the value contained by *variable* from thr Nth character to up to *length* specidied.

${*variable*/*pattern*/*string*}
The longest match of *pattern* against the *variable* value is replaced with *string*.

Print commands

echo *My home is: $HOME* Write arguments to the
>My home is: /home/*user* standard output.

echo –e Enable interpretation of backslash-escaped characters.

printf Format and print the arguments.

printf %q "$IFS" Print the arguments shell-quoted.
>' \t\n'

printf "%.1f" 2.558 Specify the decimal precision.
>2.6

printf "%s\t%s\n" "1" "2" "3" "4" %s interprets the
>1 2 associated argument
3 4 literally as string.

Using quotes

Weak quoting - double quote (") :

***string*="My home is: $HOME"**
echo $*string*
>My home is: /home/*user*
Use when you want to enclose variables or use shell expansion inside a string.

Strong quoting - single quote (') :

echo 'My HOME is: $HOME'
>My HOME is: $HOME
Preserves the literal value of each character within the quotes.

Wildcards operators

Regular expressions : Used to match text.

^	Matches the beginning of the line.
$	Matches the end of the line.
^$	Matches blank lines.
.	Any character.
[]	Any of the character inside the brackets.
[^a-f]	Matches any character except those in the range a to f.
\a	A letter (similar to [a-zA-Z]).
\t	A tabulation.
\n	A new line.
\w	An alphanumeric ([a-zA-Z0-9_])
\W	Non alphanumeric (The opposite of \w)
?	The preceding item matches 0 or 1 time.
*****	The preceding item matches 0 or more times.
+	The preceding item matches 1 or more times.
{N}	The preceding item matches exactly N times.
{N,}	The preceding item matches N times or more.
{N,M}	The preceding item matches at least N times and not more than M times.
[:*class*:]	POSIX Character Classes ([:alnum:], [:alpha:], [:blank:], [:digit:], etc, respectively equivalent to A-Za-z0-9, A-Za-z, space or a tab, 0-9, etc).

Globbing (Pathname expansion) :
Used to match filename(s).

?	Any single character
*****	Zero or more characters
[]	Specify a range. Any character of the range or none of them by using **!** inside the bracket.
{*term1*,*term2*}	Specify a list of terms separated by commas and each term must be a name or a wildcard.
{term1..term2}	Called brace expansion, this syntax expands all the terms between *term1* and *term2* (Letters or Integers).

With the **extglob** shell option enabled (check it with **shopt**) :
In the following description, a *pattern-list* is a list of one or more patterns separated by a |.

man *command* : display the *command*'s manual page

Jacques Dainat - 2015

?(*pattern-list*) Matches zero or one occurrence of the given patterns.

***(*pattern-list*)** Matches zero or more occurrences of the given patterns.

+(*pattern-list*) Matches one or more occurrences of the given patterns.

@(*pattern-list*) Matches one of the given patterns.

!(*pattern-list*) Matches anything except one of the given patterns.

/!\ Regular expressions and globbing wildcards should not be mixed up. They have different meaning.

File modification commands

tr string1 string2 < file

Replace *string1* characters occurrences within *file* by *string2* characters (where the first character in string1 is translated into the first character in string2 and so on).

sed is a non-interactive text file editor :

sed 's/*pattern1*/*pattern2*/g' *ficOrigine*

Replace **pattern1** occurrence within *file* by ***pattern2***. The **s** means « substitute » and the **g** means « global replacment » (Not only the first occurence).

-e : allows combining multiple commands (use a -e before each command).

-i : Edit files in-place. (Be carefull using that option)

sed -n *5,10*p *file*

Print lines 5 to 10.

The awk command

awk is a field-oriented pattern processing language.

awk 'BEGIN { *Initial command(s)* }
{ *by line command(s)* }
END { *final command(s)* }' *file*

$0 is an entire line.
$1 is the first field, **$2** the second, etc.

By default, fields are separated by white space. Use the **–F** option to define the input field separator (can be a regular expression).

NF Number of fields in the current record.
NR Ordinal number of the current record.
FNR Ordinal number of the current record in the current file.

-v *name*=$*var* It allows to pass the shell variable $*var* to awk command. The variable is known as *name* within the awk command.

awk '{ if (*$2 ~ pattern*) *arr*[*$0*]++} END { for (*i* in *arr*){print *$i*} }' *file*

For each line where the second field match the *pattern*, save the line as key in the associative array *arr* and increment its value. At the end print each key of the associative array. This will remove the duplicate lines that have matched.

awk 'FNR==NR{*arr*[*$4*]++;next}{ if(*$4* in *arr*)print $0 }' *file1 file2*

Print all lines of *file2* where the fourth field matches one of the third field of *file1*.

String commands together

command < file

Redirect *file* into a *command*. *File* is read as standard input instead of the terminal command.

command1 | command2

Connect the standard output of the left command to the standard input of the right command.

command1 ; command2

Separate two commands. Permit putting several commands on the same line.

Math calculation

+	Plus
+=	Plus-equal (increment variable by a constant)
-	Minus.
-=	Minus-equal (decrement variable by a constant).
*	Multiplication.
*=	Times-equal (multiply variable by a constant).
/	Division.
/=	Slash-equal (divide variable by a constant).
%	Modulo (returns the remainder of an integer division operation).
%=	Modulo-equal (remainder of dividing variable by a constant).
**	Exponentiation.
++	Increment a variable by 1.
--	Decrement a variable by 1.

((*var* = *operation*)) or ***var*=$((*operation*))**

Assign the result of an arithmetic evaluation to the variable *var*.

/!\ Natively Bash can only handle integer arithmetic.

Floating-point arithmetic

You must delegate such kind of calcul to specific command line tool as **bc**.

echo "*operation*" | bc –l

Display the result of a floating-point arithmetic.

***var*=$(echo "*operation* " | bc -l)**

Assign the floating-point arithmetic result to the variable *var*.

man *command* : display the *command*'s manual page

Bash Programming Pocket Reference

lazy dogs @ dogtown <dogtown@mare-system.de>

VERSION 2.2.16 :: 19 September 2012

Abstract

A quick cheat sheet for programmers who want to do shell scripting. This is not intended to teach bash-programming. based upon: http://www.linux-sxs.org/programming/bashcheat.html for beginners, see moar References at the end of this doc

Copyright Notice

Contents

Chapter 1

Bash

1.1 Basics

All bash scripts must tell the o/s what to use as the interpreter. The first line of any script should be:

```
#!/bin/bash
```

You must either make bash scripts executable `chmod +x filename` or invoke bash with the script as argument: `bash ./your_script.sh`

1.2 Variables and getopt - get command line options

1.2.1 Variables

Create a variable - just assign value. Variables are non-datatyped (a variable can hold strings, numbers, etc. with out being defined as such). `varname=value` Display a variable via `echo` by putting $ on the front of the name; you can assign the output of a command to a variable too:

```
display:
  echo $varname

assign:
  varname=`command1 | command2 | command3`
```

Values passed in from the command line as arguments are accessed as $# where #= the index of the variable in the array of values being passed in. This array is base 1 not base 0.

```
command var1 var2 var3 .... varX
```

$1 contains whatever var1 was, $2 contains whatever var2 was, etc.

1.2.2 Built in variables:

- $1-$N :: Stores the arguments (variables) that were passed to the shell program from the command line. >
- $? :: Stores the exit value of the last command that was executed.
- $0 :: Stores the first word of the entered command (the name of the shell program).
- $* :: Stores all the arguments that were entered on the command line ($1 $2 . . .).
- "$@" :: Stores all the arguments that were entered on the command line, individually quoted (“$1” “$2” . . .).

1.2.3 getopt - command line options

```
if [ "$1" ]; then
    # options with values: o: t: i:
    # empty options: ohu
    while getopts ohuc:t:i: opt
    do
    case $opt in
    o)

        o_commands

    ;;

    u)
        u_commands
    ;;

    t)
```

```
            t_ARGS="$OPTARG"
        ;;

        *)
            exit
        ;;


        esac
        done

    fi

    shift $((OPTIND - 1))
```

1.3 Quote Marks

Regular double quotes "like these" make the shell ignore whitespace and count it all as one argument being passed or string to use. Special characters inside are still noticed/obeyed.

Single quotes 'like this' make the interpreting shell ignore all special characters in whatever string is being passed. The back single quote marks (aka backticks) (`command`) perform a different function. They are used when you want to use the results of a command in another command. For example, if you wanted to set the value of the variable contents equal to the list of files in the current directory, you would type the following command: `contents=`ls``, the results of the ls program are put in the variable contents.

1.4 Tests / Comparisons

A command called test is used to evaluate conditional expressions, such as a if-then statement that checks the entrance/exit criteria for a loop.

```
test expression

  ... or ...

[ expression ]
```

```
USAGE:

[ expression ] && do_commands
        => do_commands if expresssion is ok
```

1.4.1 Numeric Comparisons

```
int1 -eq int2   Returns True if int1 is equal to int2.
int1 -ge int2   Returns True if int1 is greater than or equal to int2.
int1 -gt int2   Returns True if int1 is greater than int2.
int1 -le int2   Returns True if int1 is less than or equal to int2
int1 -lt int2   Returns True if int1 is less than int2
int1 -ne int2   Returns True if int1 is not equal to int2
```

1.4.2 String Comparisons

```
str1 = str2     Returns True if str1 is identical to str2.
str1 != str2    Returns True if str1 is not identical to str2.
str             Returns True if str is not null.
-n str          Returns True if the length of str is
                greater than zero.
-z str          Returns True if the length of str is equal
                to zero. (zero is different than null)
```

1.4.3 File Comparisons

```
-d filename     Returns True if filename is a directory.
-e filename     Returns True if filename exists (might be a directory
-f filename     Returns True if filename is an ordinary file.
-h filename     Returns True if filename is a symbolic link
-p filename     Returns True if filename is a pipe
-r filename     Returns True if filename can be read by the process.
-s filename     Returns True if filename has a nonzero length.
-S filename     Returns True if filename is a Socket
-w filename     Returns True if file, filename can be written by the process.
```

```
-x filename     Returns True if file, filename is executable.

$fd1 -nt $fd2   Test if fd1 is newer than fd2. The modification date is used
$fd1 -ot $fd2   Test if fd1 is older than fd2. The modification date is used
$fd1 -ef $fd2   Test if fd1 is a hard link to fd2
```

1.4.4 Expression Comparisons

```
!expression     Returns true if expression is not true
expr1 -a expr2  Returns True if expr1 and expr2 are true.
                ( && , and )
expr1 -o expr2  Returns True if expr1 or expr2 is true.
                ( ||, or )
```

1.4.5 testing if $var is an integer

src1: http://www.linuxquestions.org/questions/programming-9/test-for-integer-in-bash-279227/#post1514631

src2: http://stackoverflow.com/questions/806906/how-do-i-test-if-a-variable-is-a-number-in-bash

```
You can also use expr to ensure a variable is numeric

a=100

if [ `expr $a + 1 2> /dev/null` ] ; then
    echo $a is numeric ;
else
    echo $a is not numeric ;
fi


example 2:

[[ $1 =~ "^[0-9]+$" ]] && echo "number" && exit 0 || echo "not a number" && exit
```

1.5 Logic and Loops

1.5.1 if ... then ... elif ... else

```
-> you can always write: (( if [ expression ]; then )) as shortcut

if [ expression ]
    then
    commands
fi

if [ expression ]
    then
    commands
else
    commands
fi

if [ expression ]
    then
    commands
elif [ expression2 ]
    then
    commands
else
    commands
fi


# arithmethic in if/test

function mess {
   if (( "$1" > 0 )) ; then
      total=$1
   else
      total=100
   fi
   tail -$total /var/log/messages | less

}
```

1.5.2 Loops

it can be usefull to assign IFS_Values for your script before running and reassign default-values at the end.

```
ORIGIFS=$IFS
IFS=`echo -en " \n\b"`

for var1 in list
    do
    commands
done
IFS=$ORIGIFS
```

This executes once for each item in the list. This list can be a variable that contains several words separated by spaces (such as output from ls or cat), or it can be a list of values that is typed directly into the statement. Each time through the loop, the variable var1 is assigned the current item in the list, until the last one is reached.

```
while [ expression ]
    do
    commands
done

until [ expression ]
    do
    commands
done
```

1.5.3 Case select

```
case string1 in
    str1)
        commands1
        ;;
    str2)
```

```
        commands2
        ;;
    *)
        commands3
        ;;
esac
```

string1 is compared to str1 and str2. If one of these strings matches string1, the commands up until the double semicolon (; ;) are executed. If neither str1 nor str2 matches string1, the commands associated with the asterisk are executed. This is the default case condition because the asterisk matches all strings.

1.5.4 select -> select from a list of values

```
export PS3="
 alternate_select_prpmpt # > "
"
select article_file in $sgml_files
    do
    case $REPLY in
    x)
    exit
    ;;
    q)
    exit
    ;;
    esac
    NAME="$article_file"
    break
    done



fi
```

1.6 bash foo

1.6.1 input/output-redirection

Three le descriptors (0, 1 and 2) are automatically opened when a shell in invoked. They represent:

```
 0    standard input (stdin)
1    standard output (stdout)
2    standard error (stderr)
```

A commandâs input and output may be redirected using the following notation:

```
 <file    take input from le
>file     write output to le
          (truncate to zero if it exists)
>>file    append output to le, else create
<<word    \here" document; read input until line matches word
<>file    open le for reading and writing
<&digit   use le descriptor digit as input
          (>&digit for output)
<&-       close standard input (>&- close output)
cmd1|cmd2 stdout of cmd1 is piped to stdin of cmd2
```

```
 ls -l >listing
```

```
ls -l | lpr
zcat file.tar.Z | tar tvf -
```

1.6.2 Functions

Create a function:

```
fname(){
    commands
}
```

you can call then the function `fname`, giving $ARGS as $1 $2

1.6.3 read user input

In many ocations you may want to prompt the user for some input, and there are several ways to achive this. This is one of those ways. As a variant, you can get multiple values with read, the second example may clarify this.

```
#!/bin/bash
echo Please, enter your name
read NAME
echo "Hi $NAME!"
```

```
#!/bin/bash
echo Please, enter your firstname and lastname
read FN LN
echo "Hi! $LN, $FN !"
```

1.6.4 reading return values / outputs from commands

In bash, the return value of a program is stored in a special variable called $?. This illustrates how to capture the return value of a program, I assume that the directory dada does not exist. (This was also suggested by mike)

```
#!/bin/bash
cd /dada &> /dev/null
echo rv: $?
cd $(pwd) &> /dev/null
echo rv: $?
```

Capturing a commands output

This little scripts show all tables from all databases (assuming you got MySQL installed).

```
#!/bin/bash
DBS=`mysql -uroot  -e"show databases"`
for b in $DBS ;
do
        mysql -uroot -e"show tables from $b"
done
```

1.6.5 Arithmetic & Bash-Expansion

```
i=$(( i + 1 ))
let i+=1
i=$(( i++))
let i++


Operators:

Op  Operation with assignment   Use     Meaning
-----------------------------------------------
=   Simple assignment           a=b     a=b
*=  Multiplication              a*=b    a=(a*b)
/=  Division                    a/=b    a=(a/b)
%=  Remainder                   a%=b    a=(a%b)
+=  Addition                    a+=b    a=(a+b)
-=  Subtraction                 a-=b    a=(a-b)
```

1.6.6 using Arrays

src: http://www.softpanorama.org/Scripting/Shellorama/arithmetic_expressions.shtml

Initialization of arrays in bash has format similar to Perl:

```
solaris=(serv01 serv02 serv07 ns1 ns2)
```

Each element of the array is a separate word in the list enclosed in parentheses. Then you can refer to each this way:

```
echo solaris is installed on ${solaris[2]}
```

If you omit index writing echo $solaris you will get the first element too. Another example taken from Bash Shell Programming in Linux

```
array=(red green blue yellow magenta)

len=${#array[*]}
echo "The array has $len members. They are:"
```

```
i=0
while [ $i -lt $len ]; do
    echo "$i: ${array[$i]}"
    let i++
done
```

1.6.7 date & time - conversion

```
get date in iso-formate
    now_time=`date +%F - %H:%M:%S`

get unix_timestamp
    unix_time=`date %s`

convert unix-timestamp to iso-date
    date --date "1970-01-01 $unix_time sec" "+%Y-%m-%d %T"

date_strftime - macros / format-controls:
   %%     a literal %
   %a     localeâs abbreviated weekday name (e.g., Sun)
   %A     localeâs full weekday name (e.g., Sunday)
   %b     localeâs abbreviated month name (e.g., Jan)
   %B     localeâs full month name (e.g., January)
   %c     localeâs date and time (e.g., Thu Mar  3 23:05:25 2005)
   %C     century; like %Y, except omit last two digits (e.g., 21)
   %d     day of month (e.g, 01)
   %D     date; same as %m/%d/%y
   %e     day of month, space padded; same as %_d
   %F     full date; same as %Y-%m-%d
   %g     last two digits of year of ISO week number (see %G)
   %G     year of ISO week number (see %V); normally useful only with %V
   %h     same as %b
   %H     hour (00..23)
   %I     hour (01..12)
   %j     day of year (001..366)
   %k     hour ( 0..23)
   %l     hour ( 1..12)
   %m     month (01..12)
   %M     minute (00..59)
   %n     a newline
```

```
    %N     nanoseconds (000000000..999999999)
    %p     localeâs equivalent of either AM or PM; blank if not known
    %P     like %p, but lower case
    %r     localeâs 12-hour clock time (e.g., 11:11:04 PM)
    %R     24-hour hour and minute; same as %H:%M
    %s     seconds since 1970-01-01 00:00:00 UTC
    %S     second (00..60)
    %t     a tab
    %u     day of week (1..7); 1 is Monday
    %U     week number of year, with Sunday as first day of week (00..53)
    %V     ISO week number, with Monday as first day of week (01..53)
    %w     day of week (0..6); 0 is Sunday
    %W     week number of year, with Monday as first day of week (00..53)
    %x     localeâs date representation (e.g., 12/31/99)
    %X     localeâs time representation (e.g., 23:13:48)
    %y     last two digits of year (00..99)
    %Y     year
    %z     +hhmm numeric timezone (e.g., -0400)
    %:z    +hh:mm numeric timezone (e.g., -04:00)
    %::z   +hh:mm:ss numeric time zone (e.g., -04:00:00)
    %:::z  numeric time zone with : to necessary precision (e.g., -04, +05:30)
    %Z     alphabetic time zone abbreviation (e.g., EDT)
```

1.6.8 parsing a simple conf in bash

src: http://www.chimeric.de/blog/2007/1122_parsing_simple_config_files_in_bash

The function uses some of the more advanced bash features like parameter substitution a.s.o. which I won't explain here. For a good read on the whole bash scripting topic I recommend the Advanced Bash Scripting Guide.

```
# simple configuration file
#
# default settings
default {
    DATE_PREFIX=$(date -I)
    EXT_FULL="full"
    EXT_DIFF="diff"
    SSHFS_OPTS="-C"
    DAR_OPTS="-v -m 256 -y -s 600M -D"
    DAR_NOCOMPR="-Z '*.gz' -Z '*.bz2' -Z '*.zip' -Z '*.png'"
}
```

```
# backup target system
system {
    SRC_DIR="/"
    DEST_DIR="/mnt/data/backups/tatooine"
    PREFIX="system"
    TYPE="R"
    HOST="chi$@coruscant"
}

# backup target home
home {
    SRC_DIR="/home/user"
    DEST_DIR="/mnt/data/backups/tatooine"
    PREFIX="home-nomedia"
    TYPE="R"
    HOST="chi$@coruscant"
    DAR_EXCLUDES="media"
}

------------------

#!/usr/bin/env bash
# $@author Michael Klier chi$@chimeric.de

function readconf() {

    match=0

    while read line; do
        # skip comments
        [[ ${line:0:1} == "#" ]] && continue

        # skip empty lines
        [[ -z "$line" ]] && continue

        # still no match? lets check again
        if [ $match == 0 ]; then

            # do we have an opening tag ?
            if [[ ${line:$((${#line}-1))} == "{" ]]; then
```

```
                # strip "{"
                group=${line:0:$((${#line}-1))}
                # strip whitespace
                group=${group// /}

                # do we have a match ?
                if [[ "$group" == "$1" ]]; then
                    match=1
                    continue
                fi

                continue
            fi

        # found closing tag after config was read - exit loop
        elif [[ ${line:0} == "}" && $match == 1 ]]; then
            break

        # got a config line eval it
        else
            eval $line
        fi

    done < "$CONFIG"
}

CONFIG="/home/user/.sampleconfig"

readconf "default"

echo $DATE_PREFIX
echo $DAR_OPTS
echo $DAR_NOCOMPR
```

1.6.9 extracting filenames from path/urls:

```
url="http://www.emergingthreats.org/rules/emerging_all.rules"
rules_name=${url##*/}
# $rules_name _> emerging_all.rules
wget -O $rules_name $url
```

```
path="/var/log/some.log"
file_name=${path##*/}
```

1.6.10 extracting/deleting first/latest char from string:

src: http://blog.pregos.info/2011/10/06/bash-delete-last-character-from-string/

```
Print last char from string:

    user@desktop:~$ VAR=foobar
    user@desktop:~$ echo $VAR
    foobar
    user@desktop:~$ echo ${VAR: -1}
    r


Delete last character from string:

    user@desktop:~$ VAR=foobar
    user@desktop:~$ echo $VAR
    foobar
    user@desktop:~$ echo ${VAR%?}
    fooba


Delete first character from string

    user@desktop:~$ VAR=foobar
    user@desktop:~$ echo $VAR
    foobar
    user@desktop:~$ echo ${VAR:1}
    oobar
```

1.7 usefull Shell-Commands

stuff like awk, sed etc

1.7.1 crontab - adds from commandline

src: http://dbaspot.com/solaris/386215-adding-line-crontab-command-line.html

```
Re: Adding line in crontab from command line...

On Wed, 9 Apr 2008 11:17:47 -0700 (PDT), contracer11@gmail.com wrote:
>
> Hi:
>
> Can you tell me if is there any way to make this task ?
>
>
> 00 1 * * * /monitor_file_system 2>/dev/null > crontab
>

crontab -l | (cat;echo "00 1 * * * /monitor_file_system") | crontab

Helmut

--
Almost everything in life is easier to get into than out of.
(Agnes' Law)
```

1.7.2 sed-examples

src: http://www.grymoire.com/Unix/Sed.html

SED is a tool to manipulate text-streams (Stream EDitor), together with redirects it might be used to substitute text from/ to files.

The character after the s is the delimiter. It is conventionally a slash, because this is what ed, more, and vi use. It can be anything you want, however. If you want to change a pathname that contains a slash - say /usr/local/bin to /common/bin - you could use the backslash to quote the slash:

```
$ sed 's[delimiter]ot[delimiter]nt[delimiter]{flags} < input > output

$ sed 's/\/usr\/local\/bin/\/common\/bin/' < old >new
$ sed 's_/usr/local/bin_/common/bin_' < old >new
```

```
 $ sed 's:/usr/local/bin:/common/bin:' < old >new
 $ sed 's|/usr/local/bin|/common/bin|' < old >new

combining commands:

sed -e 's/a/A/' -e 's/b/B/' < old >new

seing multiples files (f1..3)
 $ sed 's/^#.*//' f1 f2 f3 | grep -v '^$' | wc -l

grep-simulation: Nothing is printed, except those lines with PATTERN included.
 $ sed -n 's/PATTERN/&/p' file

The simplest restriction is a line number. If you wanted to delete the
first number on line 3, just add a "3" before the command:

 $ sed '3 s/[0-9][0-9]*//' < file >new

restrict to the first 100 lines:

 $ sed '1,100 s/A/a/'

pexecute from line 101 until the end; "$"  means the last
line in the file.

 $ sed '101,$ s/A/a/'
```

Pick one you like. As long as it's not in the string you are looking for, anything goes. And remember that you need three delimiters. If you get a "Unterminated 's' command" it's because you are missing one of them.

SED-FLAGS

with no flags given the fiorst found pattern is changed.

```
    /g    -> global sustitution (every occurance) instead of the first one
    /2    -> change only the second pattern
    /2g   -> change everythiong from the 2nd pattern onwards
    /p    -> print foudn matches (sed -n .../p -> simulate grep
    /w fd -> write outpuf to file $fd
```

Chapter 2

Regular Expressions

2.1 POSIX Character Classes for Regular Expressions & their meanings

Class	Meaning
[:alpha:]	Any letter, [A-Za-z]
[:upper:]	Any uppercase letter, [A-Z]
[:lower:]	Any lowercase letter, [a-z]
[:digit:]	Any digit, [0-9]
[:alnum:]	Any alphanumeric character, [A-Za-z0-9]
[:xdigit:]	Any hexadecimal digit, [0-9A-Fa-f]
[:space:]	A tab, new line, vertical tab, form feed, carriage return, or space
[:blank:]	A space or a tab.
[:print:]	Any printable character
[:punct:]	Any punctuation character: ! ’ # S % & ’ ( ) * + , − . / : ; < = > ? @ [/] ^ _ { | } ~
[:graph:]	Any character defined as a printable character except those defined as part of the space character class
[:word:]	Continuous string of alphanumeric characters and underscores.
[:ascii:]	ASCII characters, in the range: 0-127
[:cntrl:]	Any character not part of the character classes: [:upper:], [:lower:], [:alpha:], [:digit:], [:punct:], [:graph:], [:print:], [:xdigit:]

2.2 Special Characters in Regular Expressions

```
Char      Meaning                                        Example
*         Match zero, one or more of the previous          1
?         Match zero or one of the previous                2
+         Match one or more of the previous                3
\         Used to escape a special character               4
.         Wildcard character, matches any character        5
( )       Group characters                                 6
[ ]       Matches a range of characters                    7
[0-9]+    matches any positive integer
[a-zA-Z]  matches ascii letters a-z (uppercase and lower case)
[^0-9]    matches any character not 0-9.
|         Matche previous OR next character/group          8
{ }       Matches a specified number of occurrences        9
^         Beginning of a string / set                     10
$         End of a string.                                11

Examples:
    1  -  Ah* matches "Ahhhhh" or "A"
    2  -  Ah? matches "Al" or "Ah"
    3  -  Ah+ matches "Ah" or "Ahhh" but not "A"
    4  -  Hungry\? matches "Hungry?"
    5  -  do.* matches "dog", "door", "dot", etc.
    6  -  see 8
    7  -  [cbf]ar matches "car", "bar", or "far"
    8  -  (Mon)|(Tues)day matches "Monday" or "Tuesday"
    9  -  [0-9]{3} matches "315" but not "31"
          [0-9]{2,4} matches "12", "123", and "1234"
          [0-9]{2,} matches "1234567..."
   10  -  ^http matches strings that begin with http, such as a url.
          [^0-9]    matches any character not 0-9.
   11  -  ing$ matches "exciting" but not "ingenious"
```

2.3 Usefulle RegExes

- Email: [_a-zA-Z0-9-]+(\.[_a-zA-Z0-9._-]+)*@([a-zA-Z0-9._-]+\.)+([a-zA-Z]{2,4})
- IP: /^(\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3})$/

Chapter 3

Editor - Quick References

3.1 Emacs Refernces

3.1.1 Basics

- **C-x-f** find file (open file)
- **C-x-s** save current buffer
- **C-x s** save buffers that have been altered
- **C-x-w** save as
- **C-x-c** quit

3.1.2 Help

- **C-h a cmd** Get help on command
- **C-h k** Answers 'what does this key combination do?'

3.1.3 Killing and yanking

- **C-k** kill to end of line
- **M-d** kill to end of word (adding to kill ring)
- **M-delete** back-kill to start of word (adding to kill ring)

- **C-y** yank (paste)
- **M-y** yank previous(Do C-y M-y M-y M-y to yank third last kill)
- **C-space** start mark
- **M-w** kill from mark to here
- **C-insert** copy as kill from mark to here

3.1.4 Navigating

- **C-e** goto end of current line
- **C-a** goto beginning of current line
- **M-<** goto beginning buffer
- **M->** goto end of buffer
- **M-x** goto-line RET goes to specified line
- **C-M-f** goto closing brace (standing on the opening brace)
- **C-M-b** goto opening brace (standing on the closing brace)
- **C-u C-space** which takes you back to wherever you were previously working
- **M-m** position cursor on start of indentation

3.1.5 Window/Buffer commands

- **C-x 2** Split window horizontally
- **C-x 3** Split window vertically
- **C-x 1** Close all other windows but the one where the cursor is
- **C-x 0** (Zero)Close this window, keep the other
- **C-x o** (oh!) Jump to next window
- **C-x b** View another buffer
- **C-x-b** Pick another buffer to view
- **C-l** Center buffer around line
- **C-u 1 C-l** Make this line the top line of the buffer

3.1.6 Search and replace

- **C-s** incremental search forward
- **C-r** incremental search backward
- **M-%** Query replace
- **C-M-%** Query replace regexp
- **M-x** occur Find regexp in current buffer
- **M-x** grep-find Find regexp recursively from a directory
- **M-x** occur find occurences in this file, present as a list

3.1.7 Miscellaneous

- **C-g** quit whatever command (you did something you did not intend)
- **M-** remove white space before and after cursor
- **M-^** join this line with the prevoius and fix white space
- **M-x** delete-trailing-whitespace removes blanks after last char on all lines
- **C-x C-t** transpose lines, move the current line one line upwards
- **M-t** transpose words, swap the word behind the cursor for the one after
- **M-l** make the rest of this word lower case
- **M-u** make the rest of this word lower case
- **C-x C-l** make region lower case
- **C-x C-u** make region upper case

3.1.8 Navigating code

- **M-.** Jump to tag (where does this function reside? go there!)
- **M-x** goto-line RET goes to specified line

3.2 vi pocket Reference

src: http://www.lagmonster.org/docs/vi.html

command_mode: [ESC]

3.2.1 Modes

Vi has two modes insertion mode and command mode. The editor begins in command mode, where the cursor movement and text deletion and pasting occur. Insertion mode begins upon entering an insertion or change command. [ESC] returns t he editor to command mode (where you can quit, for example by typing :q!). Most commands execute as soon as you type them except for "colon" commands which execute when you press the ruturn key.

3.2.2 File Handling

- **:w** Write file
- **:w!** Write file (ignoring warnings)
- **:w!** file Overwrite file (ignoring warnings)
- **:wq** Write file and quit
- **:q** Quit
- **:q!** Quit (even if changes not saved)
- **:w file** Write file as file, leaving original untouched
- **ZZ** Quit, only writing file if changed
- **:x** Quit, only writing file if changed
- **:n1,n2w file** Write lines n1 to n2 to file
- **:n1,n2w » file** Append lines n1 to n2 to file
- **:e file2 Edit file2** (current file becomes alternate file)
- **:e!** Reload file from disk (revert to previous saved version)
- **:e#** Edit alternate file
- **%** Display current filename

- **#** Display alternate filename
- **:n** Edit next file
- **:n!** Edit next file (ignoring warnings)
- **:n files** Specify new list of files
- **:r file** Insert file after cursor
- **:r !command** Run command, and insert output after current line

3.2.3 Quitting

- **:x** Exit, saving changes
- **:q** Exit as long as there have been no changes
- **ZZ** Exit and save changes if any have been made
- **:q!** Exit and ignore any changes

3.2.4 Inserting Text

- **i** Insert before cursor
- **I** Insert before line
- **a** Append after cursor
- **A** Append after line
- **o** Open a new line after current line
- **O** Open a new line before current line
- **r** Replace one character
- **R** Replace many characters

3.2.5 Motion

- **h** Move left
- **j** Move down
- **k** Move up
- **l** Move right
- **w** Move to next word
- **W** Move to next blank delimited word
- **b** Move to the beginning of the word
- **B** Move to the beginning of blank delimted word
- **e** Move to the end of the word
- **E** Move to the end of Blank delimited word
- **(** Move a sentance back
- **)** Move a sentance forward
- { Move a paragraph back
- } Move a paragraph forward
- **0** Move to the begining of the line
- **$** Move to the end of the line
- **1G** Move to the first line of the file
- **G** Move to the last line of the file
- **nG** Move to nth line of the file
- **:n** Move to nth line of the file
- **fc** Move forward to c
- **Fc** Move back to c
- **H** Move to top of screen
- **M** Move to middle of screen
- **L** Move to botton of screen
- **%** Move to associated ( ), { }, [ ]

3.2.6 Deleting Text

Almost all deletion commands are performed by typing d followed by a motion. For example, dw deletes a word. A few other deletes are:

- **x** Delete character to the right of cursor
- **X** Delete character to the left of cursor
- **D** Delete to the end of the line
- **dd** Delete current line
- **:d** Delete current line

3.2.7 Yanking Text

Like deletion, almost all yank commands are performed by typing y followed by a motion. For example, y$ yanks to the end of the line. Two other yank commands are:

- **yy** Yank the current line
- **:y** Yank the current line

3.2.8 Buffers

Named buffers may be specified before any deletion, change, yank or put command. The general prefix has the form “c where c is any lowercase character. for example, ”adw deletes a word into buffer a. It may thereafter be put back into text with an appropriate “ap.

3.2.9 Search for strings

- **/string** Search forward for string
- **?string** Search back for string
- **n** Search for next instance of string
- **N** Search for previous instance of string

3.2.10 Replace

The search and replace function is accomplished with the :s command. It is commonly used in combination with ranges or the :g command (below).

- **:s/pattern/string/flags** Replace pattern with string according to flags.
- **g Flag** - Replace all occurences of pattern
- **c Flag** - Confirm replaces.
- **&** Repeat last :s command

3.2.11 Regular Expressions

- **. (dot)** Any single character except newline
- * zero or more occurances of any character
- **[...]** Any single character specified in the set
- **[^...]** Any single character not specified in the set
- **^** Anchor - beginning of the line
- **$** Anchor - end of line
- **\<** Anchor - begining of word
- **\>** Anchor - end of word
- **\(...\)** Grouping - usually used to group conditions
- **\n** Contents of nth grouping
- ——————————————————————————
- **[...]** Set Examples [A-Z] The SET from Capital A to Capital Z
- **[a-z]** The SET from lowercase a to lowercase z
- **[0-9]** The SET from 0 to 9 (All numerals)
- **[./=+]** The SET containing . (dot), / (slash), =, and +
- **[-A-F]** The SET from Capital A to Capital F and the dash (dashes must be specified first)
- **[0-9 A-Z]** The SET containing all capital letters and digits and a space
- **[A-Z][a-zA-Z]** In the first position, the SET from Capital A to Capital Z In the second character position, the SET containing all letters

3.2.12 Regular Expression Examples

- **/Hello/** Matches if the line contains the value Hello
- **/^TEST$/** Matches if the line contains TEST by itself
- **/^[a-zA-Z]/** Matches if the line starts with any letter
- **/^[a-z].*/** Matches if the first character of the line is a-z and there is at least one more of any character following it
- **/2134$/** Matches if line ends with 2134
- **/\(21|35\)/** Matches is the line contains 21 or 35; Note the use of ( ) with the pipe symbol to specify the 'or' condition
- **/[0-9]*/** Matches if there are zero or more numbers in the line
- **/^[^#]/** Matches if the first character is not a # in the line

Notes:

1 Regular expressions are case sensitive

2 Regular expressions are to be used where pattern is specified

3.2.13 Counts

Nearly every command may be preceded by a number that specifies how many times it is to be performed. For example, 5dw will delete 5 words and 3fe will move the cursor forward to the 3rd occurence of the letter e. Even insertions may be repeated conveniently with thismethod, say to insert the same line 100 times.

3.2.14 Other

```
~       Toggle upp and lower case
J       Join lines
.       Repeat last text-changing command
u       Undo last change
U       Undo all changes to line
```

Chapter 4

Links and Resources

4.1 Links and Resources

- Advanced Bash Scripting guide: http://tldp.org/guides.html#abs

Arithmetic Operators

```
$ var=$(( 20 + 5 ))
$ expr 1 + 3   # 4
$ expr 2 - 1   # 1
$ expr 10 / 3  # 3
$ expr 20 % 3  # 2 (remainder)
$ expr 10 \* 3 # 30 (multiply)
```

String Operators

Expression	Meaning
`${#str}`	Length of $str
`${str:pos}`	Extract substring from $str at $pos
`${str:pos:len}`	Extract $len chars from $str at $pos
`${str/sub/rep}`	Replace first match of $sub with $rep
`${str//sub/rep}`	Replace all matches of $sub with $rep
`${str/#sub/rep}`	If $sub matches front end of $str, substitute $rep for $sub
`${str/%sub/rep}`	If $sub matches back end of $str, substitute $rep for $sub

Relational Operators

Num	String	Test
`-eq`	`=`	Equal to
	`==`	Equal to
`-ne`	`!=`	Not equal to
`-lt`	`\<`	Less than
`-le`		Less than or equal to
`-gt`	`\>`	Greater than
`-ge`		Greater than or equal to
	`-z`	is empty
	`-n`	is not empty

File Operators

	True if file exists and...
`-f file`	...is a regular file
`-r file`	...is readable
`-w file`	...is writable
`-x file`	...is executable
`-d file`	...is a directory
`-s file`	...has a size greater than zero.

Control Structures

```
if [ condition ] # true = 0
  then
# condition is true
elif [ condition1 ]
  then
# condition1 is true
 elif condition2
   then
# condition2 is true
else
# None of the conditions is true
fi

case expression in
  pattern1) execute commands ;;
  pattern2) execute commands ;;
esac

while [ true ]
do
#  execute commands
done

until [ false ]
do
#   execute commands
done

for x in 1 2 3 4 5 # or for x in {1..5}
do
    echo "The value of x is $x";
done

LIMIT=10
for ((x=1; x <= LIMIT ; x++))
do
  echo -n "$x "
done

for file in *~
do
  echo "$file"
done

break [n] # exit n levels of loop
continue [n] # go to next iteration of loop n up
```

Function Usage

```
function-name arg1 arg2 arg3 argN
```

n.b. functions must be defined before use...

Function Definition

```
function function-name ()
{
# statement1
# statement2
# statementN
  return [integer] # optional
}
```

Functions have access to script variables, and may have local variables:

```
$ local var=value
```

Arrays

```
$ vars[2]=”two” # declare an array
$ echo ${vars[2]} # access an element
$ fruits=(apples oranges pears) # populate array
$ echo ${fruits[0]} # apples - index from 0
$ declare -a fruits # creates an array

echo "Enter your favourite fruits: "
read -a fruits
echo You entered ${#fruits[@]} fruits
for f in "${fruits[@]}"
do
  echo "$f"
done

$ array=( "${fruits[@]}" "grapes" ) # add to end
$ copy="${fruits[@]}" # copy an array
$ unset fruits[1] # delete one element
$ unset fruits # delete array
```

Array elements do not have to be sequential - indices are listed in `{!fruits[@]}`:

```
for i in ${!fruits[@]}
do
  echo fruits[$i]=${fruits[i]}
done
```

All variables are single element arrays:

```
$ var=”The quick brown fox”
$ echo {var[0]} #  The quick brown fox
```

String operators can be applied to all the string elements in an array using ${name[@] ... } notation, e.g.:

```
$ echo ${arrayZ[@]//abc/xyz} # Replace all
occurrences of abc with xyz
```

User Interaction

```
echo -n "Prompt: "
read
echo "You typed $REPLY."

echo -n "Prompt: "
read response
echo "You typed $response."

PS3="Choose a fruit: "
select fruit in "apples" "oranges" "pears"
do
  if [ -n "$fruit" ]
  then
    break
  fi
  echo "Invalid choice"
done

$ dialog --menu "Choose" 10 20 4 1 apples 2 \
oranges 3 pears 4 bananas 2>/tmp/ans
$ fruit=`cat /tmp/ans`
$ echo $fruit

$ zenity --list --radiolist --column "Choose" \
--column "Fruit" 0 Apples 0 Oranges 0 Pears 0 \
Bananas > /tmp/ans
$ fruit=`cat /tmp/ans`
$ echo $fruit
```

Reading Input from a File

```
exec 6<&0              # 'Park' stdin on #6
exec < temp.txt        # stdin=file "temp.txt"
read                   # from stdin
until [ -z "$REPLY" ]
do
  echo "$REPLY"        # lists temp.txt
  read
done
exec 0<&6 6<&-         # restore stdin
echo -n "Press any key to continue"
read
```

Trapping Exceptions

```
TMPFILE=`mktemp`
on_break()
{
  rm -f $TMPFILE
  exit 1
}
trap on_break 2 # catches Ctrl+C
```

Data and Time

```
$ start=`date +%s`
$ end=`date +%s`
$ echo That took $((end-start)) seconds
$ date +"%c" -d19540409
Fri 09 Apr 1954 12:00:00 AM GMT
```

Case Conversion

```
$ in="The quick brown fox"
$ out=`echo $in | tr [:lower:] [:upper:]`
$ echo "$out"
THE QUICK BROWN FOX
```

Preset Variables

`$HOME`	User's home directory
`$HOSTNAME`	Name of host
`$HOSTTYPE`	Type of host (e.g. i486)
`$PWD`	Current directory
`$REPLY`	default variable for READ and SELECT
`$SECONDS`	Elapsed time of script
`$TMOUT`	Max. script elapsed time or wait time for read

References

Linux Shell Scripting Tutorial - A Beginner's handbook
http://www.cyberciti.biz/nixcraft/linux/docs/uniqlinuxfeatures/lsst/
BASH Programming Introduction, Mike G
http://www.tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html
Advanced BASH Scripting Guide, Mendel Cooper
http://tldp.org/LDP/abs/html/

Copyright & Licence

This version dated:

BASH Quick Reference Card

"All the useful stuff on a single card"

```
#!/bin/bash
$ chmod ugo+x shell_script.sh

$ bash [options] [file]
```

Options
-x show execution of [file]
-v echo lines as they are read

Variables

```
$ var="some value" # declare a variable
$ echo $var # access contents of variable
$ echo ${var} # access contents of variable
$ echo ${var:-"default value"} # with default
$ var= # delete a variable
$ unset var # delete a variable
```

Quoting - "`$variable`" - preserves whitespace

Positional Variables

`$0`	Name of script
`$1-$9`	Positional parameters #1 - #9
`${10}`	to access positional parameter #10 onwards
`$#`	Number of positional parameters
`"$*"`	All the positional parameters (as a single word) *
`"$@"`	All the positional parameters (as separate strings)
`$?`	Return value

`set [values]` - sets positional params to `[values]`
`set --` - deletes all positional parameters
`shift [n]`- move positional params n places to the left

Command Substitution

```
$ var=`ls *.txt` # Variable contains output
$ var=$(ls *.txt)   # Alternative form
$ cat myfile >/dev/null # suppress stdout
$ rm nofile 2>/dev/null # suppress stderr
$ cat nofile 2>/dev/null >/dev/null # suppress
both
```

Bash-Scripting Cheat-Sheet

for confidently writing shell-scripts with the bash

Bash-Script
Cheat-Sheet

... use shebang line as first line of your script ...

```
#!/bin/bash
... your code goes here ...
```

... to prevent users from running your script with the wrong interpreter

... start your script the right way ...

exact: copy to $PATH and make executable

```
#> cp script.sh /usr/local/bin
#> chod +x /usr/local/bin/script.sh
#> script.sh
#> Hello World!
```

fast: start your script with bash

```
#> bash script.sh
Hello World!
```

... defining and using variables ...

set with NAME=Value

```
#> NAME=John
#> FULLNAME="John Doe"
#> COUNTER=0
#> export TZ=asia/tokyo
```

use value with $NAME

```
#> echo Hello $FULLNAME
Hello John Doe
#> PATH=$PATH:~/bin
#> export TZ=asia/tokyo date
```

***** don't forget to export variables, which are to be used by subsequent commands *****

... define variables as permanent ...

*** variables don't just exist "somewhere in the os" ***
they have to be defined within a process and are inherited by the it's sub-processes if exported

		login-shell	**other shell**
for all users	➜	/etc/profile	/etc/bash.bashrc
for one user	➜	~/.profile or ~./bash_profile	~/.bashrc

** if bash is running as a script, it doesn't read any of the files above **

Bash-Script
Cheat-Sheet

... quoting the right way ...

** three ways to help the bash to **not interprete special characters** **

\x	...	do not interprete the one character after the \
“abc”	...	do not interprete any special character except $
‘abc ’	...	interprete nothing within the quotes

... working with variables ...

strings: no special voodoo needed

```
#> FIRST=John
#> LAST=Doe
#> FULL=”$FIRST $LAST”
#> echo Hello $FULL
Hello John Doe
```

calculating: use $((...)) syntax

```
#> A=5; B=13
#> SUM=$(( $A + $B ))
#> echo $(( $B / $A ))
2
```

... special variables ...

$? ...returncode of last command
$$ ... PID of actual shell/script
$! ... PID of last background process

$1..$9 ... positional parameters
use “shift” if you need more than 9 parameters

... use “source” or “.” to **include** other files ...

config.sh

```
LOGFILE=/var/log/debug.log
LOGTAG=DEBUG
```

script.sh

```
. config.sh
echo “$LOGTAG hi log” >> $LOGFILE
```

... use command-output just like variables ...

** surround a command by $(...) to use it’s output directly on the command-line **

example1

```
#> echo “Hello, I am $(whoami)”
Hello, I am john
```

example1

```
#> cp file file_$(date +%Y_%m_%d)
#> ls
file file_2017_10_31
```

Bash-Script
Cheat-Sheet

... make use of $? for implementing logic ...

cmd1 && cmd2 ➜ cmd2 only gets executed, if cmd1 returns 0 ($? == 0)
cmd1 || cmd2 ➜ cmd2 only gets executed, if cmd1 not returns 0 ($? != 0)

** chain as many commands, as you need **

... pre-define your truth :-) ...

```
#> which true
/bin/true
#> true
#> echo $?
0
```

```
#> which false
/bin/false
#> false
#> echo $?
1
```

... set your own return-code ...

by using “exit N”, you exit the script and set the return-code to N
N can be in the range from 0 to 255

... negate every return-code ...

** prepend a command with an exclamation mark, to negate it’s returncode **

```
#> ! true
#> echo $?
1
```

```
#> ! false
#> echo $?
0
```

... “test” sets $? to show results for your tests ...

compare strings / numbers

test string = string	➜ equal
test string != string	➜ not equal
test number -eq number	➜ equal
test number -ne number	➜ not equal
(other: -lt, -le, -ge, -ge)	➜ (lower than, …)

files

test -x file	➜ file exists?
test -r file	➜ file readable?
test -w file	➜ file writeable?
test -d dir	➜ is dir a directory?
test file1 -nt / -ot file2	➜ file1 older/newer than file 2

** test <expression> ist equivalent to [<expression>] **mind the spaces around [and]** **

Bash-Script
Cheat-Sheet

... conditional logic ...

if - then - else

```
if test-command
then
        cmd
        …
else
        cmd
        …
fi
```

- runs test-command
- if it returns 0 ($?), run cmds between then end else
- else run commands between else and fi
- else-block is optional

case

```
case $VAR in
        test1 )     cmd
                    …
                    ;;
        test2 )     cmd
                    …
                    ;;
        …
esac
```

- tests content of variable VAR
- tests are written as strings (e.g. “yes” “no”) or as search-patterns (e.g. “[Yy]*” or “[Nn]*”)
- only the commands of the first match are run

... mostly used loops ...

for-loop

```
for VAR in E1 E2 E3 E4 ...
do
        cmd
        …
done
```

- runs the commands between **do** and **done** for every element in the list after **in**
- while the commands are run, **VAR** contains the element for whitch the loop is run.

while-loop

```
while test-command
do
        cmd
        ...
done
```

- runs test-command
- if test-command returns “0” ($?), run the commands between **do** and **done**
- starts again with test-command and continues until test-command does not return 0.

Bash-Script
Cheat-Sheet

... redirecting streams ...

**** every command leverages per default three data-streams ****

default:

stdin is read from console / keyboard
stdout and stderr are bound to console
stdout and stderr are mixed

connecting commands:

cmd1 | cmd2 ➜ connect stdout of cmd1 to stdin of cmd2

working with files:

cmd > file	➜ write **stdout** to new generated file
cmd >> file	➜ append **stdout** to file
cmd 2> file	➜ write **stderr** to new generated file
cmd 2>> file	➜ append **stderr** to file
cmd < file	➜ read **stdin** from file
cmd 2>&1	➜ connect stderr and stdout both to stdout

Tip: use xargs to copy stdout to next command-line as parameters: #> ls | xargs rm

I hope you enjoyed ....

This cheat-sheet is meant to be a quick overview over fundamental concepts of bash-scripting (for some of you a review and new stuff for others).

If anything doesn’t make sense or is confusing - don’t worry.

I’ll be back soon with great material to make it all crystal-clear :-)

Bash scripting *cheatsheet*

Example

```
#!/usr/bin/env bash

NAME="John"
echo "Hello $NAME!"
```

Variables

```
NAME="John"
echo $NAME
echo "$NAME"
echo "${NAME}!"
```

String quotes

```
NAME="John"
echo "Hi $NAME"  #=> Hi John
echo 'Hi $NAME'  #=> Hi $NAME
```

Shell execution

```
echo "I'm in $(pwd)"
echo "I'm in `pwd`"
# Same
```

See Command substitution

Conditional execution

```
git commit && git push
git commit || echo "Commit failed"
```

Functions

```
get_name() {
  echo "John"
}

echo "You are $(get_name)"
```

See: Functions

Conditionals

```
if [[ -z "$string" ]]; then
  echo "String is empty"
elif [[ -n "$string" ]]; then
```

```
  echo "String is not empty"
fi
```

See: Conditionals

Strict mode

```
set -euo pipefail
IFS=$'\n\t'
```

See: Unofficial bash strict mode

Brace expansion

```
echo {A,B}.js
```

`{A,B}`	Same as `A B`
`{A,B}.js`	Same as `A.js B.js`
`{1..5}`	Same as `1 2 3 4 5`

See: Brace expansion

#Parameter expansions

Basics

```
name="John"
echo ${name}
echo ${name/J/j}    #=> "john" (substitution)
echo ${name:0:2}    #=> "Jo" (slicing)
echo ${name::2}     #=> "Jo" (slicing)
echo ${name::-1}    #=> "Joh" (slicing)
echo ${name:(-1)}   #=> "n" (slicing from right)
echo ${name:(-2):1} #=> "h" (slicing from right)
echo ${food:-Cake}  #=> $food or "Cake"

length=2
echo ${name:0:length}  #=> "Jo"
```

See: Parameter expansion

```
STR="/path/to/foo.cpp"
echo ${STR%.cpp}    # /path/to/foo
echo ${STR%.cpp}.o  # /path/to/foo.o

echo ${STR##*.}     # cpp (extension)
echo ${STR##*/}     # foo.cpp (basepath)

echo ${STR#*/}      # path/to/foo.cpp
echo ${STR##*/}     # foo.cpp

echo ${STR/foo/bar} # /path/to/bar.cpp

STR="Hello world"
echo ${STR:6:5}   # "world"
echo ${STR:-5:5}  # "world"

SRC="/path/to/foo.cpp"
```

```
BASE=${SRC##*/}   #=> "foo.cpp" (basepath)
DIR=${SRC%$BASE}  #=> "/path/to/" (dirpath)
```

Substitution

`${FOO%suffix}`	Remove suffix
`${FOO#prefix}`	Remove prefix
`${FOO%%suffix}`	Remove long suffix
`${FOO##prefix}`	Remove long prefix
`${FOO/from/to}`	Replace first match
`${FOO//from/to}`	Replace all
`${FOO/%from/to}`	Replace suffix
`${FOO/#from/to}`	Replace prefix

Comments

```
# Single line comment

: '
This is a
multi line
comment
'
```

Substrings

`${FOO:0:3}`	Substring *(position, length)*
`${FOO:-3:3}`	Substring from the right

Length

`${#FOO}` Length of `$FOO`

Manipulation

```
STR="HELLO WORLD!"
echo ${STR,}   #=> "hELLO WORLD!" (lowercase 1st letter)
echo ${STR,,}  #=> "hello world!" (all lowercase)

STR="hello world!"
echo ${STR^}   #=> "Hello world!" (uppercase 1st letter)
echo ${STR^^}  #=> "HELLO WORLD!" (all uppercase)
```

Default values

`${FOO:-val}`	`$FOO`, or `val` if not set
`${FOO:=val}`	Set `$FOO` to `val` if not set
`${FOO:+val}`	`val` if `$FOO` is set
`${FOO:?message}`	Show error message and exit if `$FOO` is not set

The `:` is optional (eg, `${FOO=word}` works)

#Loops

Basic for loop

```
for i in /etc/rc.*; do
  echo $i
done
```

C-like for loop

```
for ((i = 0 ; i < 100 ; i++)); do
  echo $i
done
```

Ranges

```
for i in {1..5}; do
    echo "Welcome $i"
done
```

With step size

```
for i in {5..50..5}; do
    echo "Welcome $i"
done
```

Reading lines

```
< file.txt | while read line; do
  echo $line
done
```

Forever

```
while true; do
  ...
done
```

#Functions

Defining functions

```
myfunc() {
    echo "hello $1"
}

# Same as above (alternate syntax)
function myfunc() {
    echo "hello $1"
}

myfunc "John"
```

Returning values

```
myfunc() {
```

```
    local myresult='some value'
    echo $myresult
}

result="$(myfunc)"
```

Raising errors

```
myfunc() {
  return 1
}

if myfunc; then
  echo "success"
else
  echo "failure"
fi
```

Arguments

`$#`	Number of arguments
`$*`	All arguments
`$@`	All arguments, starting from first
`$1`	First argument

See Special parameters.

#Conditionals

Conditions

Note that `[[` is actually a command/program that returns either `0` (true) or `1` (false). Any program that obeys the same logic (like all base utils, such as `grep(1)` or `ping(1)`) can be used as condition, see examples.

`[[ -z STRING ]]`	Empty string
`[[ -n STRING ]]`	Not empty string
`[[ STRING == STRING ]]`	Equal
`[[ STRING != STRING ]]`	Not Equal
`[[ NUM -eq NUM ]]`	Equal
`[[ NUM -ne NUM ]]`	Not equal
`[[ NUM -lt NUM ]]`	Less than
`[[ NUM -le NUM ]]`	Less than or equal
`[[ NUM -gt NUM ]]`	Greater than
`[[ NUM -ge NUM ]]`	Greater than or equal
`[[ STRING =~ STRING ]]`	Regexp
`(( NUM < NUM ))`	Numeric conditions
`[[ -o noclobber ]]`	If OPTIONNAME is enabled
`[[ ! EXPR ]]`	Not
`[[ X ]] && [[ Y ]]`	And
`[[ X ]] || [[ Y ]]`	Or

File conditions

`[[ -e FILE ]]`	Exists
`[[ -r FILE ]]`	Readable
`[[ -h FILE ]]`	Symlink
`[[ -d FILE ]]`	Directory
`[[ -w FILE ]]`	Writable
`[[ -s FILE ]]`	Size is > 0 bytes
`[[ -f FILE ]]`	File
`[[ -x FILE ]]`	Executable
`[[ FILE1 -nt FILE2 ]]`	1 is more recent than 2
`[[ FILE1 -ot FILE2 ]]`	2 is more recent than 1
`[[ FILE1 -ef FILE2 ]]`	Same files

Example

```
# String
if [[ -z "$string" ]]; then
  echo "String is empty"
elif [[ -n "$string" ]]; then
  echo "String is not empty"
fi

# Combinations
if [[ X ]] && [[ Y ]]; then
  ...
fi

# Equal
if [[ "$A" == "$B" ]]

# Regex
if [[ "A" =~ "." ]]

if (( $a < $b )); then
   echo "$a is smaller than $b"
fi

if [[ -e "file.txt" ]]; then
  echo "file exists"
fi
```

#Arrays

Defining arrays

```
Fruits=('Apple' 'Banana' 'Orange')

Fruits[0]="Apple"
Fruits[1]="Banana"
Fruits[2]="Orange"
```

Working with arrays

```
echo ${Fruits[0]}           # Element #0
```

```
echo ${Fruits[@]}           # All elements, space-separated
echo ${#Fruits[@]}          # Number of elements
echo ${#Fruits}             # String length of the 1st element
echo ${#Fruits[3]}          # String length of the Nth element
echo ${Fruits[@]:3:2}       # Range (from position 3, length 2)
```

Operations

```
Fruits=("${Fruits[@]}" "Watermelon")    # Push
Fruits+=('Watermelon')                  # Also Push
Fruits=( ${Fruits[@]/Ap*/} )            # Remove by regex match
unset Fruits[2]                         # Remove one item
Fruits=("${Fruits[@]}")                 # Duplicate
Fruits=("${Fruits[@]}" "${Veggies[@]}") # Concatenate
lines=(`cat "logfile"`)                 # Read from file
```

Iteration

```
for i in "${arrayName[@]}"; do
  echo $i
done
```

#Dictionaries

Defining

```
declare -A sounds

sounds[dog]="bark"
sounds[cow]="moo"
sounds[bird]="tweet"
sounds[wolf]="howl"
```

Declares sound as a Dictionary object (aka associative array).

Working with dictionaries

```
echo ${sounds[dog]} # Dog's sound
echo ${sounds[@]}   # All values
echo ${!sounds[@]}  # All keys
echo ${#sounds[@]}  # Number of elements
unset sounds[dog]   # Delete dog
```

Iteration

Iterate over values

```
for val in "${sounds[@]}"; do
  echo $val
done
```

Iterate over keys

```
for key in "${!sounds[@]}"; do
  echo $key
done
```

#Options

Options

```
set -o noclobber  # Avoid overlay files (echo "hi" > foo)
set -o errexit    # Used to exit upon error, avoiding cascading errors
set -o pipefail   # Unveils hidden failures
set -o nounset    # Exposes unset variables
```

Glob options

```
set -o nullglob    # Non-matching globs are removed  ('*.foo' => '')
set -o failglob    # Non-matching globs throw errors
set -o nocaseglob  # Case insensitive globs
set -o globdots    # Wildcards match dotfiles ("*.sh" => ".foo.sh")
set -o globstar    # Allow ** for recursive matches ('lib/**/*.rb' =>
'lib/a/b/c.rb')
```

Set `GLOBIGNORE` as a colon-separated list of patterns to be removed from glob matches.

#History

Commands

`history`	Show history
`shopt -s histverify`	Don’t execute expanded result immediately

Expansions

`!$`	Expand last parameter of most recent command
`!*`	Expand all parameters of most recent command
`!-n`	Expand `n`th most recent command
`!n`	Expand `n`th command in history
`!<command>`	Expand most recent invocation of command `<command>`

Operations

`!!`	Execute last command again
`!!:s/<FROM>/<TO>/`	Replace first occurrence of `<FROM>` to `<TO>` in most recent command
`!!:gs/<FROM>/<TO>/`	Replace all occurrences of `<FROM>` to `<TO>` in most recent command
`!$:t`	Expand only basename from last parameter of most recent command
`!$:h`	Expand only directory from last parameter of most recent command

`!!` and `!$` can be replaced with any valid expansion.

Slices

`!!:n`	Expand only `n`th token from most recent command (command is `0`; first argument is `1`)
`!^`	Expand first argument from most recent command
`!$`	Expand last token from most recent command
`!!:n-m`	Expand range of tokens from most recent command
`!!:n-$`	Expand `n`th token to last from most recent command

`!!` can be replaced with any valid expansion i.e. `!cat`, `!-2`, `!42`, etc.

#Miscellaneous

Numeric calculations

```
$((a + 200))      # Add 200 to $a

$((RANDOM%=200))  # Random number 0..200
```

Subshells

```
(cd somedir; echo "I'm now in $PWD")
pwd # still in first directory
```

Redirection

```
python hello.py > output.txt   # stdout to (file)
python hello.py >> output.txt  # stdout to (file), append
python hello.py 2> error.log   # stderr to (file)
python hello.py 2>&1           # stderr to stdout
python hello.py 2>/dev/null    # stderr to (null)
python hello.py &>/dev/null    # stdout and stderr to (null)

python hello.py < foo.txt      # feed foo.txt to stdin for python
```

Inspecting commands

```
command -V cd
#=> "cd is a function/alias/whatever"
```

Trap errors

```
trap 'echo Error at about $LINENO' ERR
```

or

```
traperr() {
  echo "ERROR: ${BASH_SOURCE[1]} at about ${BASH_LINENO[0]}"
}

set -o errtrace
trap traperr ERR
```

Case/switch

```
case "$1" in
  start | up)
    vagrant up
    ;;

  *)
    echo "Usage: $0 {start|stop|ssh}"
    ;;
esac
```

Source relative

```
source "${0%/*}/../share/foo.sh"
```

printf

```
printf "Hello %s, I'm %s" Sven Olga
#=> "Hello Sven, I'm Olga

printf "1 + 1 = %d" 2
#=> "1 + 1 = 2"

printf "This is how you print a float: %f" 2
#=> "This is how you print a float: 2.000000"
```

Directory of script

```
DIR="${0%/*}"
```

Getting options

```
while [[ "$1" =~ ^- && ! "$1" == "--" ]]; do case $1 in
  -V | --version )
    echo $version
    exit
    ;;
  -s | --string )
    shift; string=$1
    ;;
  -f | --flag )
    flag=1
    ;;
esac; shift; done
if [[ "$1" == '--' ]]; then shift; fi
```

Heredoc

```
cat <<END
hello world
END
```

Reading input

```
echo -n "Proceed? [y/n]: "
read ans
echo $ans

read -n 1 ans    # Just one character
```

Special variables

`$?` Exit status of last task

`$!` PID of last background task

`$$` PID of shell

`$0` Filename of the shell script

See Special parameters.

Go to previous directory

```
pwd # /home/user/foo
cd bar/
pwd # /home/user/foo/bar
cd -
pwd # /home/user/foo
```

Check for command's result

```
if ping -c 1 google.com; then
  echo "It appears you have a working internet connection"
fi
```

Grep check

```
if grep -q 'foo' ~/.bash_history; then
  echo "You appear to have typed 'foo' in the past"
fi
```

#Also see

- Bash-hackers wiki *(bash-hackers.org)*
- Shell vars *(bash-hackers.org)*
- Learn bash in y minutes *(learnxinyminutes.com)*
- Bash Guide *(mywiki.wooledge.org)*
- ShellCheck *(shellcheck.net)*

DEVHINTS.IO Edit

Bash scripting cheatsheet

Example

```
#!/usr/bin/env bash

NAME="John"
echo "Hello $NAME!"
```

Variables

```
NAME="John"
echo $NAME
echo "$NAME"
echo "${NAME}!"
```

String quotes

```
NAME="John"
echo "Hi $NAME"  #=> Hi John
echo 'Hi $NAME'  #=> Hi $NAME
```

Shell execution

```
echo "I'm in $(pwd)"
echo "I'm in `pwd`"
# Same
```

Command substitution

See Command substitution

```
echo {A,B}.js
```

`{A,B}`	Same as `A B`
`{A,B}.js`	Same as `A.js B.js`
`{1..5}`	Same as `1 2 3 4 5`

See: Brace expansion

Parameter expansions

Basics

```
name="John"
echo ${name}
echo ${name/J/j}    #=> "john" (subst
echo ${name:0:2}    #=> "Jo" (slicing
echo ${name::2}     #=> "Jo" (slicing
echo ${name::-1}    #=> "Joh" (slicin
echo ${name:(-1)}   #=> "n" (slicing
echo ${name:(-2):1} #=> "h" (slicing
echo ${food:-Cake}  #=> $food or "Cak
```

```
length=2
echo ${name:0:length}  #=> "Jo"
```

See: Parameter expansion

```
STR="/path/to/foo.cpp"
echo ${STR%.cpp}    # /path/to/foo
echo ${STR%.cpp}.o  # /path/to/foo.o

echo ${STR##*.}     # cpp (extension)
echo ${STR##*/}     # foo.cpp (basepa

echo ${STR#*/}      # path/to/foo.cpp
echo ${STR##*/}     # foo.cpp

echo ${STR/foo/bar} # /path/to/bar.cp
```

```
STR="Hello world"
echo ${STR:6:5}   # "world"
```

`${FOO:-val}`	$FOO, or val if not set
`${FOO:=val}`	Set $FOO to val if not set
`${FOO:+val}`	val if $FOO is set
`${FOO:?message}`	Show error message and exit if $FOO is not set

The `:` is optional (eg, `${FOO=word}` works)

`${FOO/%from/to}`	Replace suffix
`${FOO/#from/to}`	Replace prefix

♯ Loops

Basic for loop

```
for i in /etc/rc.*; do
  echo $i
done
```

C-like for loop

```
for ((i = 0 ; i < 100 ; i++)); do
```

```
  echo $i
done
```

Ranges

Reading lines

```
while true; do
  ...
done
```

With step size

```
for i in {5..50..5}; do
    echo "Welcome $i"
done
```

Functions

Defining functions

```
myfunc() {
    echo "hello $1"
}
```

```
# Same as above (alternate syntax)
function myfunc() {
```

```
echo "hello $1"
}
```

$#	Number of arguments
$*	All arguments
$@	All arguments, starting from first
$1	First argument

See Special parameters.

Conditionals

Conditions

Note that `[[` is actually a command/program that returns either `0` (true) or `1` (false). Any program that obeys the same logic (like all base utils, such as `grep(1)` or `ping(1)`) can be used as condition, see examples.

`[[ -z STRING ]]`	Empty string
`[[ -n STRING ]]`	Not empty string
`[[ STRING == STRING ]]`	Equal
`[[ STRING != STRING ]]`	Not Equal
`[[ NUM -eq NUM ]]`	Equal
`[[ NUM -ne NUM ]]`	Not equal
`[[ NUM -lt NUM ]]`	Less than
`[[ NUM -le NUM ]]`	Less than or equal
`[[ NUM -gt NUM ]]`	Greater than
`[[ NUM -ge NUM ]]`	Greater than or equal
`[[ STRING =~ STRING ]]`	Regexp

File conditions

`[[ -e FILE ]]`	Exists
`[[ -r FILE ]]`	Readable

```
# String
if [[ -z "$string" ]]; then
  echo "String is empty"
elif [[ -n "$string" ]]; then
  echo "String is not empty"
fi
```

```
# Combinations
if [[ X ]] && [[ Y ]]; then
  ...
fi
```

```
# Equal
if [[ "$A" == "$B" ]]
```

```
# Regex
if [[ "A" =~ "." ]]
```

```
if (( $a < $b )); then
   echo "$a is smaller than $b"
fi
```

```
if [[ -e "file.txt" ]]; then
  echo "file exists"
fi
```

♯ Arrays

Defining arrays

```
Fruits=('Apple' 'Banana' 'Orange')
```

```
Fruits[0]="Apple"
Fruits[1]="Banana"
Fruits[2]="Orange"
```

Working with arrays

```
echo ${Fruits[0]}           # Element
echo ${Fruits[@]}           # All ele
echo ${#Fruits[@]}          # Number
echo ${#Fruits}             # String
echo ${#Fruits[3]}          # String
echo ${Fruits[@]:3:2}       # Range (
```

Operations

```
Fruits=("${Fruits[@]}" "Watermelon")
Fruits+=('Watermelon')
Fruits=( ${Fruits[@]/Ap*/} )
unset Fruits[2]
Fruits=("${Fruits[@]}")
Fruits=("${Fruits[@]}" "${Veggies[@]}
lines=(`cat "logfile"`)
```

Iteration

```
for i in "${arrayName[@]}"; do
  echo $i
done
```

Dictionaries

Defining

```
declare -A sounds
```

```
sounds[dog]="bark"
sounds[cow]="moo"
sounds[bird]="tweet"
sounds[wolf]="howl"
```

Declares sound as a Dictionary object (aka associative array).

Working with dictionaries

```
echo ${sounds[dog]} # Dog's sound
echo ${sounds[@]}   # All values
echo ${!sounds[@]}  # All keys
echo ${#sounds[@]}  # Number of eleme
unset sounds[dog]   # Delete dog
```

Iteration

Iterate over values

```
for val in "${sounds[@]}"; do
  echo $val
done
```

Iterate over keys

```
for key in "${!sounds[@]}"; do
  echo $key
done
```

Options

Globbing options

```
set -o nullglob    # Non-matching glo
set -o failglob    # Non-matching glo
set -o nocaseglob  # Case insensitive
set -o globdots    # Wildcards match
set -o globstar    # Allow ** for rec
```

Set `GLOBIGNORE` as a colon-separated list of patterns to be removed from glob matches.

History

Commands

`history`	Show history
`shopt -s histverify`	Don't execute expanded result immediately

Expansions

`!$`	Expand last parameter of most recent command
`!*`	Expand all parameters of most recent command
`!-n`	Expand nth most recent command

Slices

`!!`	Execute last command
`!!:n`	Expand only nth token from most recent command (command is `0`; first argument is `1`)
`!^`	Expand first argument from most recent command
`!$`	Expand last token from most recent command
`!!:n-m`	Expand range of tokens from most recent command
`!!:n-$`	Expand nth token to last from most recent command

`!!` can be replaced with any valid expansion i.e. `!cat`, `!-2`, `!42`, etc.

`!$:h`	Expand only directory from last parameter of most recent command

`!!` and `!$` can be replaced with any valid expansion.

Miscellaneous

Numeric calculations

```
$((a + 200))      # Add 200 to $a
```

```
$((RANDOM%=200))"  # Random number 0..
```

```
if grep -q 'foo' ~/.bash_history; the
  echo "You appear to have typed 'foo
fi
```

```
pwd # /home/user/foo
```

`$0`	Filename of the shell script

See Special parameters.

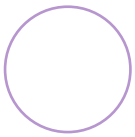
```
    ;;
esac; shift; done
if [[ "$1" == '--' ]]; then shift; fi
```

♯ Also see

- Bash-hackers wiki (bash-hackers.org)
- Shell vars (bash-hackers.org)
- Learn bash in y minutes (learnxinyminutes.com)
- Bash Guide (mywiki.wooledge.org)
- ShellCheck (shellcheck.net)

▶ **17 Comments** for this cheatsheet. Write yours!

devhints.io / Search 381+ cheatsheets



Over 381 curated cheatsheets, by developers for developers.

Devhints home

Other CLI cheatsheets

- Cron cheatsheet
- Homebrew cheatsheet
- httpie cheatsheet
- adb (Android Debug Bridge) cheatsheet
- composer cheatsheet
- Fish shell cheatsheet

Top cheatsheets

- Elixir cheatsheet
- ES2015+ cheatsheet
- React.js cheatsheet
- Vimdiff cheatsheet
- Vim cheatsheet
- Vim scripting cheatsheet

v1.3

steve-parker.org #!

Sept 2012

Shell Scripting Cheat Sheet for Unix and Linux

Online: http://steve-parker.org/sh/sh.shtml
Book: http://steve-parker.org/shellscripting

File Redirection

> file	create (overwrite) file
>> file	append to file
< file	read from file
a | b	Pipe 'a' as input to 'b'

Common Constructs

$ while read f > do > echo “Line is $f” > done < file	read text file line by line note: “$” prompt becomes “>”
$ grep foo myfile afoo foo foobar	find lines in myfile containing the text “foo”
$ cut -d: -f5 /etc/passwd Steve Parker	get 5th field delimited by colon
$ cmd1 || cmd2	run cmd1; if fails, run cmd2
$ cmd1 && cmd2	run cmd1; if it works, run cmd2
case $foo in a) echo “foo is A” ;; b) echo “foo is B” ;; *) echo “foo is not A or B” ;; esac	act upon the value of a variable note that “;;” is required at the end of each section
myvar=`ls`	get output of ls into variable

doubleit() {
expr $1 * 2
}
doubleit 3 # returns 6
	function declaration and syntax for calling it

Test Operators

```
if [ “$x” -lt “$y” ]; then
  # do something
fi
```

Numeric Tests

lt	less than
gt	greater than
eq	equal to
ne	not equal
ge	greater or equal
le	less or equal

File Tests

nt	newer than
d	is a directory
f	is a file
x	executable
r	readable
w	writeable

String Tests

=	equal to
z	zero length
n	not zero length

Logical Tests

&&	logical AND
||	logical OR
!	logical NOT

Arguments

$0	program name
$1	1st argument
$2	2nd argument
…	…
$#	no. of arguments
$*	all arguments

Variable Substitution

${V:-default}	$V, or “default” if unset
${V:=default}	$V (set to “default” if unset)
${V:?err}	$V, or “err” if unset

Conditional Execution

cmd1 || cmd2	run cmd1; if fails, run cmd2
cmd1 && cmd2	run cmd1; if ok, run cmd2

Files

mv /src /dest	move /src into /dest
ls a*	list files beginning with “a”
ls *a	list files ending with “a”
ls -ltr	list oldest first, newest last
ls -lSr	list smallest first, biggest last
ls -a	list all files, including hidden
find /src -print \ | cpio -pudvm	copy /src into current directory, preserving links, special devices, etc.

Preset Variables

$SHELL	what shell am I running?
$RANDOM	provides random numbers
$$	PID of current process
$?	return code from last cmd
$!	PID of last background cmd

Generally Useful Commands

file /etc/hosts	determine file type
basename /bin/ls	strip directory name (ls)
dirname /bin/ls	get directory name (/bin)
ifconfig -a	show all network adapters
netstat -r	show routers
netstat -a	show open ports
date +%Y%m%d	Year, Month, Day
date +%H%M	Hours, Minutes
wc -l	count number of lines
pwd	present working directory

Misc Useful Commands and Tools

egrep “(foo|bar)” file	find “foo” or “bar” in file
awk '{ print $5 }' file	print the 5th word of each line
cal 3 1973	March 1973
df -h	show disk mounts
three=`expr 1 + 2`	simple maths
echo "scale = 5 ; \ 5121 / 1024" | bc	better maths (5.00097)
time cmd	stopwatch on cmd
touch file	create blank file
alias ll='ls -l'	alias for ls -l
unalias ls	unset existing alias

find . -size 10k -print	files over 10Kb
find . -name “*.txt” -print	find text files
find /foo -type d -ls	list all directories under /foo
less file	display file page by page
sed s/foo/bar/g file	replace “foo” with “bar”
sed -i s/foo/bar/g file	in file (-i: update file)
strace -tfp PID	trace system calls for PID
tar cvf archive.tar file1 file 2 file3	create tar archive
ssh user@host	log in to host as user
scp file.txt user@host:	copy file.txt to host as user
scp user@host:/tmp/file.txt /var/tmp	copy /tmp/file.txt from user at host to /var/tmp locally
cd -	return to previous directory

Cheatography

Linux Command Line Cheat Sheet

by DaveChild

Bash Commands

uname -a	Show system and kernel
head -n1 /etc/issue	Show distribution
mount	Show mounted filesystems
date	Show system date
uptime	Show uptime
whoami	Show your username
man *command*	Show manual for *command*

Bash Shortcuts

CTRL-c	Stop current command
CTRL-z	Sleep program
CTRL-a	Go to start of line
CTRL-e	Go to end of line
CTRL-u	Cut from start of line
CTRL-k	Cut to end of line
CTRL-r	Search history
!!	Repeat last command
!*abc*	Run last command starting with *abc*
!*abc*:p	Print last command starting with *abc*
!$	Last argument of previous command
!*	All arguments of previous command
^*abc*^*123*	Run previous command, replacing *abc* with *123*

Bash Variables

env	Show environment variables
echo *$NAME*	Output value of *$NAME* variable
export *NAME=value*	Set *$NAME* to *value*
$PATH	Executable search path
$HOME	Home directory
$SHELL	Current shell

IO Redirection

command < *file*	Read input of *command* from *file*
command > *file*	Write output of *command* to *file*
command > /dev/null	Discard output of *command*
command >> *file*	Append output to *file*
command1 | *command2*	Pipe output of *command1* to *command2*

Directory Operations

pwd	Show current directory
mkdir *dir*	Make directory *dir*
cd *dir*	Change directory to *dir*
cd ..	Go up a directory
ls	List files

ls Options

-a	Show all (including hidden)
-R	Recursive list
-r	Reverse order
-t	Sort by last modified
-S	Sort by file size
-l	Long listing format
-1	One file per line
-m	Comma-separated output
-Q	Quoted output

Search Files

grep *pattern files*	Search for *pattern* in *files*
grep -i	Case insensitive search
grep -r	Recursive search
grep -v	Inverted search
find */dir/* -name *name**	Find files starting with *name* in *dir*
find */dir/* -user *name*	Find files owned by *name* in *dir*
find */dir/* -mmin *num*	Find files modifed less than *num* minutes ago in *dir*
whereis *command*	Find binary / source / manual for *command*
locate *file*	Find *file* (quick search of system index)

File Operations

touch *file1*	Create *file1*
cat *file1 file2*	Concatenate files and output
less *file1*	View and paginate *file1*
file *file1*	Get type of *file1*
cp *file1 file2*	Copy *file1* to *file2*
mv *file1 file2*	Move *file1* to *file2*
rm *file1*	Delete *file1*
head *file1*	Show first 10 lines of *file1*
tail *file1*	Show last 10 lines of *file1*
tail -f *file1*	Output last lines of *file1* as it changes

Process Management

ps	Show snapshot of processes
top	Show real time processes
kill *pid*	Kill process with id *pid*
pkill *name*	Kill process with name *name*
killall *name*	Kill all processes with names beginning *name*

Nano Shortcuts

Files	
Ctrl-R	Read file
Ctrl-O	Save file
Ctrl-X	Close file
Cut and Paste	
ALT-A	Start marking text
CTRL-K	Cut marked text or line
CTRL-U	Paste text
Navigate File	
ALT-/	End of file
CTRL-A	Beginning of line
CTRL-E	End of line
CTRL-C	Show line number
CTRL-_	Go to line number
Search File	
CTRL-W	Find
ALT-W	Find next
CTRL-\	Search and replace

More nano info at: http://www.nano-editor.org/docs.php

Screen Shortcuts

screen	Start a screen session.
screen -r	Resume a screen session.
screen -list	Show your current screen sessions.
CTRL-A	Activate commands for screen.
CTRL-A c	Create a new instance of terminal.
CTRL-A n	Go to the next instance of terminal.
CTRL-A p	Go to the previous instance of terminal.
CTRL-A "	Show current instances of terminals.
CTRL-A A	Rename the current instance of terminal.

File Permissions

chmod 775 *file*	Change mode of *file* to 775
chmod -R 600 *folder*	Recursively chmod *folder* to 600
chown *user*:*group* *file*	Change *file* owner to *user* and group to *group*

File Permission Numbers

The first digit is the owner permission, the second the group and the third for everyone.

Calculate each of the three permission digits by adding the numeric values of the permissions below.

4	read (r)
2	write (w)
1	execute (x)

Bash Cheat Sheet

By John Stowers

This file contains short tables of commonly used items in this shell. In most cases the information applies to both the Bourne shell (sh) and the newer bash shell.

Tests (for ifs and loops) are done with [ ] or with the test command.

Checking files:

```
-r file     Check if file is readable.
-w file     Check if file is writable.
-x file     Check if we have execute access to file.
-f file     Check if file is an ordinary file (as opposed to a directory, a device special file, etc.)
-s file     Check if file has size greater than 0.
-d file     Check if file is a directory.
-e file     Check if file exists.  Is true even if file is a directory.
```

Example:

```
if [ -s file ]
then
    #such and such
fi
```

Checking strings:

```
s1 = s2     Check if s1 equals s2.
s1 != s2    Check if s1 is not equal to s2.
-z s1       Check if s1 has size 0.
-n s1       Check if s2 has nonzero size.
s1          Check if s1 is not the empty string.
```

Example:

```
if [ $myvar = "hello" ] ; then
echo "We have a match"
fi
```

Checking numbers:
Note that a shell variable could contain a string that represents a number. If you want to check the numerical value use one of the following:

```
n1 -eq n2      Check to see if n1 equals n2.
n1 -ne n2      Check to see if n1 is not equal to n2.
n1 -lt n2      Check to see if n1 < n2.
n1 -le n2      Check to see if n1 <= n2.
n1 -gt n2      Check to see if n1 > n2.
n1 -ge n2      Check to see if n1 >= n2.
```

Example:

```
if [ $# -gt 1 ]
then
    echo "ERROR: should have 0 or 1 command-line parameters"
fi
```

Boolean operators:

```
!     not
-a    and
-o    or
```

Example:

```
if [ $num -lt 10 -o $num -gt 100 ]
then
    echo "Number $num is out of range"
elif [ ! -w $filename ]
```

```
then
    echo "Cannot write to $filename"
fi
```

Note that ifs can be nested. For example:

```
if [ $myvar = "y" ]
then
    echo "Enter count of number of items"
    read num
    if [ $num -le 0 ]
    then
        echo "Invalid count of $num was given"
    else
#... do whatever ...
fi
fi
```

The above example also illustrates the use of read to read a string from the keyboard and place it into a shell variable. Also note that most UNIX commands return a true (nonzero) or false (0) in the shell variable status to indicate whether they succeeded or not. This return value can be checked. At the command line echo $status. In a shell script use something like this:

```
if grep -q shell bshellref
then
    echo "true"
else
    echo "false"
fi
```

Note that -q is the quiet version of grep. It just checks whether it is true that the string shell occurs in the file bshellref. It does not print the matching lines like grep would otherwise do.

I/O Redirection:

```
pgm > file     Output of pgm is redirected to file.
pgm < file     Program pgm reads its input from file.
pgm >> file    Output of pgm is appended to file.
pgm1 | pgm2    Output of pgm1 is piped into pgm2 as the input to pgm2.
n > file       Output from stream with descriptor n redirected to file.
n >> file      Output from stream with descriptor n appended to file.
n >& m         Merge output from stream n with stream m.
n <& m         Merge input from stream n with stream m.
<< tag         Standard input comes from here through next tag at start of line.
```

Note that file descriptor 0 is normally standard input, 1 is standard output, and 2 is standard error output.

Shell Built-in Variables:

```
$0             Name of this shell script itself.
$1             Value of first command line parameter (similarly $2, $3, etc)
$#             In a shell script, the number of command line parameters.
$*             All of the command line parameters.
$-             Options given to the shell.
$?             Return the exit status of the last command.
$$             Process id of script (really id of the shell running the script)
```

Pattern Matching:

```
*              Matches 0 or more characters.
?              Matches 1 character.
[AaBbCc]       Example: matches any 1 char from the list.
[^RGB]         Example: matches any 1 char not in the list.
[a-g]          Example: matches any 1 char from this range.
```

Quoting:

```
\c             Take character c literally.
`cmd`          Run cmd and replace it in the line of code with its output.
"whatever"     Take whatever literally, after first interpreting $, `...`, \
'whatever'     Take whatever absolutely literally.
```

Example:

```
match=`ls *.bak`        #Puts names of .bak files into shell variable match.
echo \*                 #Echos * to screen, not all filename as in:  echo *
echo '$1$2hello'        #Writes literally $1$2hello on screen.
echo "$1$2hello"        #Writes value of parameters 1 and 2 and string hello.
```

Grouping:
Parentheses may be used for grouping, but must be preceded by backslashes since parentheses normally have a different meaning to the shell (namely to run a command or commands in a subshell). For example, you might use:

```
if test \( -r $file1 -a -r $file2 \) -o \( -r $1 -a -r $2 \)
then
    #do whatever
fi
```

Case statement:
Here is an example that looks for a match with one of the characters a, b, c. If $1 fails to match these, it always matches the * case. A case statement can also use more advanced pattern matching.

```
case "$1" in
    a) cmd1 ;;
    b) cmd2 ;;
    c) cmd3 ;;
    *) cmd4 ;;
esac
```

Loops:
Bash supports loops written in a number of forms,

```
for arg in [list]
do
    echo $arg
done

for arg in [list] ; do
    echo $arg
done
```

You can supply `[list]` directly

```
NUMBERS="1 2 3"
for number in `echo $NUMBERS`
do
  echo $number
done

for number in $NUMBERS
do
  echo -n $number
done

for number in 1 2 3
do
  echo -n $number
done
```

If `[list]` is a glob pattern then bash can expand it directly, for example:

```
for file in *.tar.gz
do
    tar -xzf $file
done
```

You can also execute statements for `[list]`, for example:

```
for x in `ls -tr *.log`
do
   cat $x &gt;&gt; biglog
done
```

Shell Arithmetic:
In the original Bourne shell arithmetic is done using the expr command as in:

```
result=`expr $1 + 2`
result2=`expr $2 + $1 / 2`
result=`expr $2 \* 5`               #note the \ on the * symbol
```

With bash, an expression is normally enclosed using [ ] and can use the following operators, in order of precedence:

```
* / %       (times, divide, remainder)
+ -         (add, subtract)
< > <= >=   (the obvious comparison operators)
== !=       (equal to, not equal to)
&&          (logical and)
||          (logical or)
=           (assignment)
```

Arithmetic is done using long integers.
Example:

```
result=$[$1 + 3]
```

In this example we take the value of the first parameter, add 3, and place the sum into result.

Order of Interpretation:
The bash shell carries out its various types of interpretation for each line in the following order:

```
brace expansion         (see a reference book)
~ expansion             (for login ids)
parameters              (such as $1)
variables               (such as $var)
command substitution    (Example:  match=`grep DNS *` )
arithmetic              (from left to right)
word splitting
pathname expansion      (using *, ?, and [abc] )
```

Other Shell Features:

```
$var           Value of shell variable var.
${var}abc      Example: value of shell variable var with string abc appended.
#              At start of line, indicates a comment.
var=value      Assign the string value to shell variable var.
cmd1 && cmd2   Run cmd1, then if cmd1 successful run cmd2, otherwise skip.
cmd1 || cmd2   Run cmd1, then if cmd1 not successful run cmd2, otherwise skip.
cmd1; cmd2     Do cmd1 and then cmd2.
cmd1 & cmd2    Do cmd1, start cmd2 without waiting for cmd1 to finish.
(cmds)         Run cmds (commands) in a subshell.
```