Chapter 4: Basic Shell Programming

Bash has some of the most advanced programming capabilities of any command interpreter of its type.

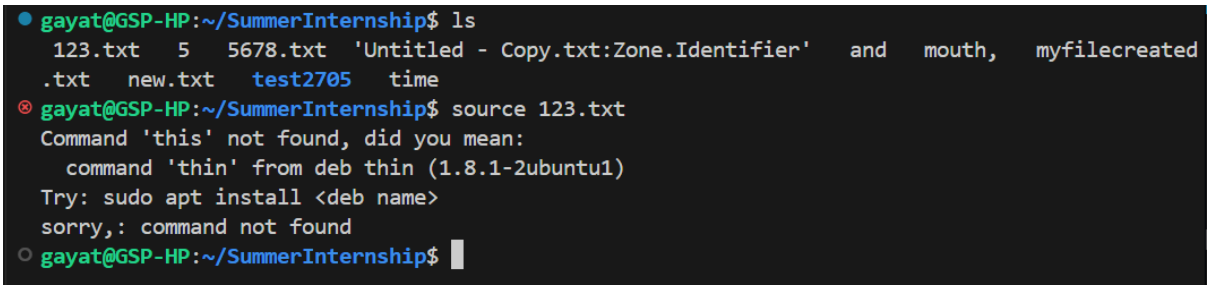
Shell Scripts and Functions:

Script- A shell program. Eg, .bash-profile, .bashrc, .bash_logout

A text file with a series of commands that are executed by an interpreter, here the shell.

To run a script:

1. **Source** scriptname
   In the example, since the file I attached as a script, it isn't working as a command. Yet you could see how, the bash is trying to run the texts like that.



2. Type the name and enter key- when you are located in the same directory.

   Files have 3 types of permission- read, write and execute and are applied to 3 categories of users- the file's owner, a group of users and everyone else.
   To give explicit permission to execute the script with execute permission, the command used is

   $chmod +x scriptname

***While using source causes the commands in the script to be run as if they were part of your login session, the "just the name" method causes the shell to do a series of things. First, it runs another copy of the shell as a subprocess; this is called a subshell. The subshell then takes commands from the script, runs them, and terminates, handing control back to the parent shell. While the source command causes the script to be run on the same shell. Using an & sign at the end of the just the file's name, you make it a background job, which is also a subprocess. Only difference is that, in case of background jobs, you can still work on the terminal while the script is executed as a background job.**

Method	Subshell?	Affects Parent Shell?	Can I use Terminal While Running?
source script.sh	No	Yes	No
./script.sh	Yes	No- as both are working as a subprocess	No
./script.sh &	Yes		Yes

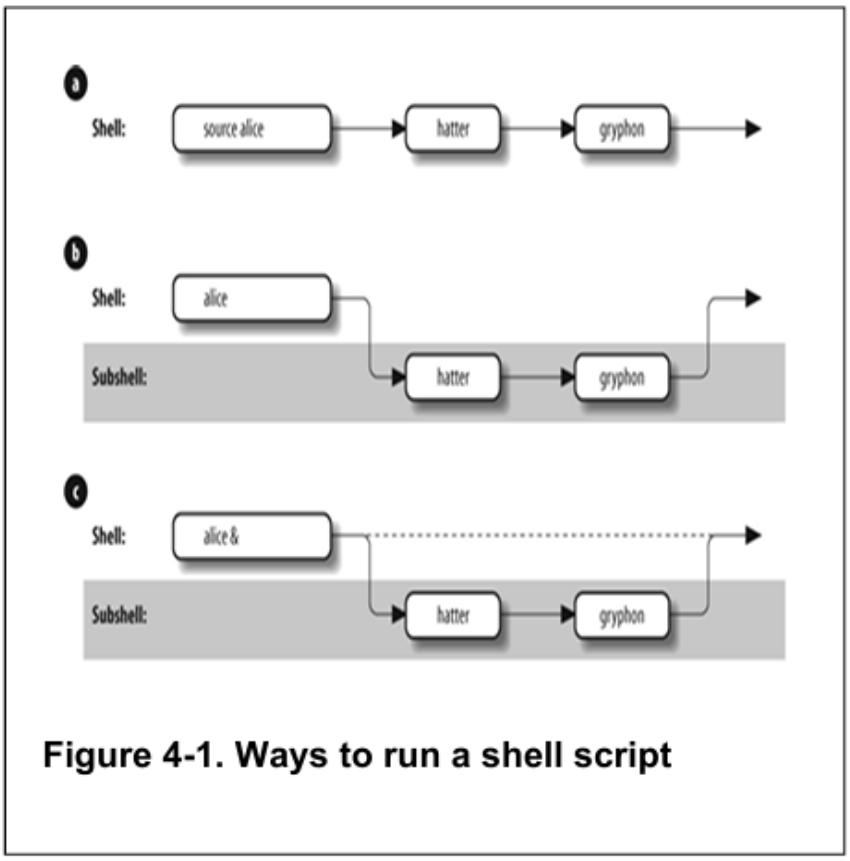


Figure 4-1. Ways to run a shell script

FUNCTIONS: 'A script within a script'

A set of commands as code put together under a name, so that every time you need it to run, just use the name of the code already written before. *Like a written recipe for a dish, so that you don't have to check for each step every time.*

Advantages of using functions: ***enhances the programmability of the shell***

- Uses the shell memory when you invoke a function
- Helpful for organising and running bulky and complex commands without typing them, and makes the terminal look neat.

HOW TO DEFINE A FUNCTION:

```
function functname
    {
        command 1
        command 2

    }
```

(less common one) OR

```
functname()
    {
        command1
        command2
    }
```

Example:

When you define a function, you tell the shell to store its name and definition (i.e., the shell commands it contains) in memory.

2 ways to run a function in Bash.

1. Write and define the function on the terminal itself and enter. Then type greet_user with the user's name, and then you get the output
2. Write the function and save it as a file and the command source for the file and then type greet_user with user's name and then you get the output

```
gayat@GSP-HP:~$ greet_user ()
    {
        echo "hello, $1! This is a trial for Fucntions!"
    }

gayat@GSP-HP:~$ greet_user gayat
hello, gayat! This is a trial for Fucntions!
gayat@GSP-HP:~$ source greet_user\ \(\).sh
gayat@GSP-HP:~$ greet_user gayat
hello, gayat! This is a trial for Fucntions!
gayat@GSP-HP:~$
```

How to unset a function: ***unset -f functname***

```
gayat@GSP-HP:~$ unset -f greet_user
gayat@GSP-HP:~$ greet_user gayat
Command 'greet_user' not found, did you mean:
  command 'greet-user' from snap greet-user (0.1)
See 'snap info <snapname>' for additional versions.
gayat@GSP-HP:~$
```

To see all the functions set with details, in alphabetical order : ***declare -f***

Just the names in alphabetical order- ***declare -F (examples next page)***

```
gayat@GSP-HP:~$ declare -f
__expand_tilde_by_ref ()
{
    if [[ ${!1-} == \~* ]]; then
        eval $1="$(printf ~%q "${!1#\~}")";
    fi
}
__get_cword_at_cursor_by_ref ()
{
    local cword words=();
    __reassemble_comp_words_by_ref "$1" words cword;
    local i cur="" index=$COMP_POINT lead=${COMP_LINE:0:COMP_POINT};
    if [[ $index -gt 0 && ( -n $lead && -n ${lead//[[:space:]]/} ) ]]; th
en
        cur=$COMP_LINE;
        for ((i = 0; i <= cword; ++i))
        do
            while [[ ${#cur} -ge ${#words[i]} && ${cur:0:${#words[i]}} !=
 "${words[i]-}" ]]; do
                cur="${cur:1}";
                ((index > 0)) && ((index--));
            done;
            if ((i < cword)); then
                local old_size=${#cur};
                cur="${cur#"${words[i]}"}";
                local new_size=${#cur};
                ((index -= old_size - new_size));
            fi;
        done;
        [[ -n $cur && ! -n ${cur//[[:space:]]/} ]] && cur=;
        ((index < 0)) && index=0;
```

declare -f

```
gayat@GSP-HP:~$ declare -F
declare -f __expand_tilde_by_ref
declare -f __get_cword_at_cursor_by_ref
declare -f __git_eread
declare -f __git_ps1
declare -f __git_ps1_colorize_gitstring
declare -f __git_ps1_show_upstream
declare -f __git_sequencer_status
declare -f __load_completion
declare -f __ltrim_colon_completions
declare -f __parse_options
declare -f __reassemble_comp_words_by_ref
declare -f _allowed_groups
declare -f _allowed_users
declare -f _available_interfaces
declare -f _bashcomp_try_faketty
declare -f _cd
declare -f _cd_devices
declare -f _command
declare -f _command_offset
declare -f _complete_as_root
declare -f _completion_loader
declare -f _configured_interfaces
declare -f _count_args
declare -f _dvd_devices
declare -f _expand
```

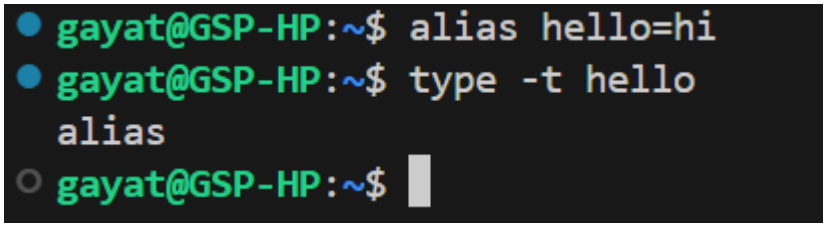
declare -F

The order of precedence for various sources of commands:

Aliases→Keywords(like function, if, else)→ functions→Built-in commands like cd, pwd→scripts and executable programs.

Precedence	Source Type	Example
1	Alias	alias ll='ls -la'
2	Keywords	if, for, function, then
3	Functions	mycmd() { echo "From function"; }
4	Built-in commands	cd, echo, type, read
5	External scripts/programs	/usr/bin/ls, ./mycmd.sh

Using the command type -all name, it gives the full definition of the alias, function or script. Type -t, gives one word definition.



SHELL VARIABLES:

Recap:

-Variables are marked by $ before the variable name.

-Environment variables are capital and known values with the export statement to subprocesses.

-Bash's high emphasis on character strings.

POSITIONAL PARAMETERS:

Special built-in variables. These hold the command-line arguments to scripts when they are invoked. Names are numbers like 1,2,3, etc, i.e. $1,$2.$4 etc.

0 is the positional character for the script invoked by the user.

The 2 special variables containing all the positional parameters, excluding 0, are * and @

"$*"→ treats all positional parameters as a single string

"$@"→treats each positional parameter as a separate quoted string. ("$@" is equal to "$1" "$2"... "$ N", where it's equal to N separate double-quoted strings)

```
gayat@GSP-HP:~$ set hello this is testing for "positional paramete
rs"
gayat@GSP-HP:~$ for arg in "$@"; do
    echo "[$arg]"
 done
[hello]
[this]
[is]
[testing]
[for]
[positional parameters]
gayat@GSP-HP:~$ for arg "$*"; do
    echo "[$arg]"
  done
bash: syntax error near unexpected token `"$*"'
[positional parameters]
bash: syntax error near unexpected token `done'
gayat@GSP-HP:~$ for arg in "$*"; do
    echo "[$arg]"
  done
[hello this is testing for positional parameters]
gayat@GSP-HP:~$ for arg in "$@"; do
    echo "[$@]"
  done
[hello this is testing for positional parameters]
[hello this is testing for positional parameters]
[hello this is testing for positional parameters]
[hello this is testing for positional parameters]
gayat@GSP-HP:~$ for arg in "$*"; do     echo "[$*]";   done
[hello this is testing for positional parameters]
gayat@GSP-HP:~$ echo "Number of args: $#"
Number of args: 6
gayat@GSP-HP:~$
```

Here, we set a line, and that gives us six parameters; the double quotes combine the last two words into one parameter. I tried four cases, the first two being the correct way of writing, which is:

for arg in “$@”; do

echo “[$arg]” ..... same for $*

done

for the $@ case, you get 6 parameters separately and for the $* case, you them as one single string!

But in the 3rd and 4th case, where $@ and $* in the place of $arg inside the square brackets.

For the "$*"→ same effect as 2nd case

For the "$@"→ the string gets printed together but 6 times-→ here the terminal cut off the last 2.

Using the command echo "Number of args: $#" you will get to see how many args are present in the statement, and that many will be shown when $@ is used.

POSITIONAL PARAMETERS IN FUNCTIONS

Global and local variables:

Global: visible everywhere in the script; defined outside the function, and can be used in the function too

Local: defined within a function and exists only when the function is executed.

```
gayat@GSP-HP:~$  function afunc
 {
 echo in function: $0 $1 $2
 var1="in function"
 echo var1: $var1
 }
 var1="outside function"
 echo var1: $var1
 echo $0: $1 $2
 afunc funcarg1 funcarg2
 echo var1: $var1
 echo $0: $1 $2
var1: outside function
/bin/bash: hello this
in function: /bin/bash funcarg1 funcarg2
var1: in function
var1: in function
/bin/bash: hello this
gayat@GSP-HP:~$
```

So basically, first the function is defined, in which, command echo $0 $1 $2 is given, where $0 gives the script name or path, others the arguments, being funcarg1 and funcarg2. These

variables are defined inside the function. Those defined outside are printed the same way. Check the output, and it is clear.

The function afunc changes the value of the variable var1 from "outside function" to "in function," and that change is known outside the function, while $1 and $2 have different values in the function and the main script. Notice that $0 doesn't change because the function executes in the environment of the shell script, and $0 takes the name of the script.

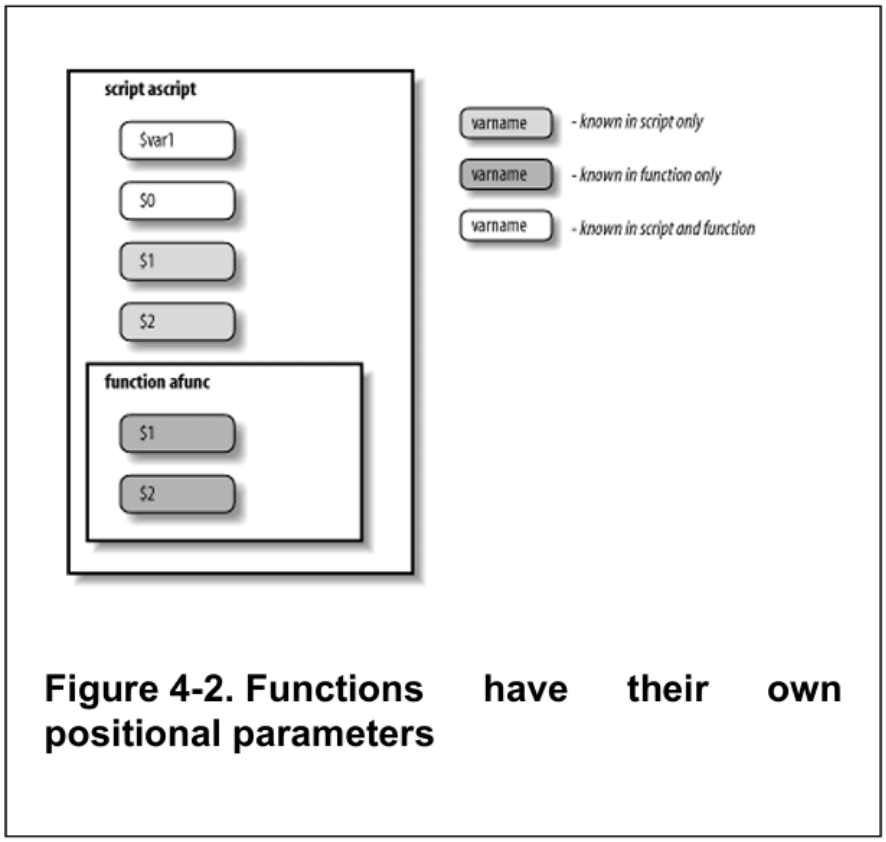


Figure 4-2. Functions have their own positional parameters

Local Variables in Functions:

A local statement inside a function definition makes the variables involved all become local to that function.

"The variable defined outside is produced first ... then, using the echo command, the function afunc is printed, which is defined within the brackets"

- ✔ First, you print the global variable.
- ✔ Then you call a function (afunc) that:
  - Echoes its parameters
  - Changes the global variable
  - Prints the updated variable

Also, afunc now has its own, local copy of var1, although the original var1 would still be used by any other functions that ascript invokes.

Quoting with $@ and $*

What is IFS-→ Internal Field Separator—a shell variable to define how bash recognises word boundaries while performing word splitting.

They are space, TAB and newline

$* separated by the first character of IFS and $@ acts like a separate double-quoted string.

Variable Syntax!

So, *$varname* is the simpler form of *${varname}!*

If you have more than 9 parameters, you need to use the latter syntax.

Similarly, variables with an underscore in their name (as in usernames) also use the latter form of syntax.

It is safe to omit the curly brackets ({}) if the variable name is followed by a character that isn't a letter, digit, or underscore.

String Operators: Allow for the manipulation of variable values without writing the whole program.

{} allows the string operators. The basic idea behind the syntax of string operators is that special characters that denote operations are inserted between the variable's name and the right curly bracket.

Their features are:

•Ensure that variables exist (i.e., are defined and have non-null values)

• Set default values for variables

• Catch errors that result from variables not being set

• Remove portions of variables' values that match patterns.

Syntax	**Meaning**
${var:-default}	Use default **if var is unset or empty** (but don't assign it) and default is set to be 0
${var:=default}	Same as above, **but also assign** default to var if it's unset/null

Syntax	Meaning
${var:+}	Testing for the existence of a variable.
${var:?error}	Exit with error if var is unset or null

```
gayat@GSP-HP:~$ count=5
sort -nr musician.sh | head -n "${count:-10}"

 5        Depeche Mode
 3        Simple Minds
 2        Split Enz
 1        Vivaldi, Antonio
```

Here, I used the sort command to sort them and then redirected it using a pipe to a head command. The head command displays the first n lines of the input. Then using a parameter expansion in bash using the value of count if it is set and not null and it's not set, use default 10. Here, I set the count=5.

```
gayat@GSP-HP:~$ sort musician.sh | head -n "${count:-10}"

 1        Vivaldi, Antonio
 2        Split Enz
 3        Simple Minds
 5        Depeche Mode
gayat@GSP-HP:~$ sort -nr musician.sh | head -n "${count:-10}"
 5        Depeche Mode
 3        Simple Minds
 2        Split Enz
 1        Vivaldi, Antonio
```

Using sort -nr: get it numerically arranged from highest to lowest

Just sort: get it arranged alphabetically.

PATTERNS AND PATTERN MATCHING

Patterns are strings with wildcard characters

The classic use for pattern-matching operators is in stripping off components of pathnames, such as directory prefixes and filename suffixes.

${variable#pattern}→if pattern matches the variable value's beginning, delete the shortest part matching the pattern

${variable##pattern}→if pattern matches the variable value's beginning, delete the longest part matching the pattern

${variable%pattern}→if pattern matches the variable value's end, delete the shortest part matching the pattern

${variable%%pattern}→if pattern matches the variable value's end, delete the longest part matching the pattern

(#-→ match the value beginning; %-→ match the value end!)

```
gayat@GSP-HP:~/SummerInternship/test2705$ path=~/SummerInternship/test2
705
gayat@GSP-HP:~/SummerInternship/test2705$ cd
gayat@GSP-HP:~$ echo $path
/home/gayat/SummerInternship/test2705
gayat@GSP-HP:~$ cd "${path##*/}"
bash: cd: test2705: No such file or directory
gayat@GSP-HP:~$ cd S
SummerInternship/ snap/
gayat@GSP-HP:~$ cd SummerInternship/
gayat@GSP-HP:~/SummerInternship$ cd "${path##*/}"
gayat@GSP-HP:~/SummerInternship/test2705$ cd "${path%*/}"
gayat@GSP-HP:~/SummerInternship/test2705$ cd "${path%%*/}"
gayat@GSP-HP:~/SummerInternship/test2705$
```

Here, I assigned the path variable with the pathname of a directory, later I tried to run the pattern parameter expansion do these. Note, in the last 2 steps, it worked as when you do %, it removes the entire pathname, leaving nothing, so cd nothing means you stay in your current directory.

```
gayat@GSP-HP:~/SummerInternship/test2705$ filename=1.pcx
gayat@GSP-HP:~/SummerInternship/test2705$ echo $filename
1.pcx
gayat@GSP-HP:~/SummerInternship/test2705$ newname="${filename%.pcx}.jpeg"
gayat@GSP-HP:~/SummerInternship/test2705$ echo $newname
1.jpeg
gayat@GSP-HP:~/SummerInternship/test2705$
```

```
gayat@GSP-HP:~/SummerInternship$ pathname=/home/gayat/SummerInternship/5
gayat@GSP-HP:~/SummerInternship$ echo $pathname
/home/gayat/SummerInternship/5
gayat@GSP-HP:~/SummerInternship$ bannername="${pathname##*/}
> "
gayat@GSP-HP:~/SummerInternship$ bannername="${pathname##*/}"
gayat@GSP-HP:~/SummerInternship$ echo bannername
bannername
gayat@GSP-HP:~/SummerInternship$ echo $bannername
5
gayat@GSP-HP:~/SummerInternship$ bannername="${pathname##/}"
gayat@GSP-HP:~/SummerInternship$ echo $bannername
home/gayat/SummerInternship/5
gayat@GSP-HP:~/SummerInternship$
```

Note, I tried 2 ways, one with the * and one without *. With the *, we are actually performing wildcard matching, allowing the shell to match up to the last slash (/). So, ##*/ means remove everything up to the last /. Without the *, you don't remove everything upto the last slash.

```
gayat@GSP-HP:~/SummerInternship$ printf "%s\n" ${PATH//:/}
/home/gayat/.vscode-server/bin/258e40fedc6cb8edf399a463ce3a9d32e7e1f6f3/bin/remote-cli/usr/loc
al/sbin/usr/local/bin/usr/sbin/usr/bin/sbin/bin/usr/games/usr/local/games/usr/lib/wsl/lib/mnt/
c/windows/system32/mnt/c/windows/mnt/c/windows/System32/Wbem/mnt/c/windows/System32/WindowsPow
erShell/v1.0//mnt/c/windows/System32/OpenSSH//mnt/c/Program
Files/HP/HP
One
Agent/mnt/c/WINDOWS/system32/mnt/c/WINDOWS/mnt/c/WINDOWS/System32/Wbem/mnt/c/WINDOWS/System32/
WindowsPowerShell/v1.0//mnt/c/WINDOWS/System32/OpenSSH//mnt/c/Users/gayat/AppData/Local/Micros
oft/WindowsApps/mnt/c/Users/gayat/AppData/Local/Programs/MiKTeX/miktex/bin/x64//mnt/c/Users/ga
yat/AppData/Local/Programs/Microsoft
VS
Code/bin/snap/bin
gayat@GSP-HP:~/SummerInternship$
```

Part	Meaning
${...}	Parameter expansion
PATH	The variable you're operating on (your list of directories)
//	Replace all occurrences (not just the first one)
:	The pattern to match (colons separate paths in $PATH)
(space)	The replacement (in this case, a single space)

LENGTH OPERATOR!

${#variablename}-→ gives the length of the value of the variable as a character string!

```
gayat@GSP-HP:~$ varname=Length_Operator
gayat@GSP-HP:~$ echo ${#varname}
15
gayat@GSP-HP:~$
```

EXTENDED PATTERN MATCHING:

extglob- 'extended globbing'- lets you use advance wildcard.

Pattern	**Example**
`@(pattern ...)`	Match **exactly one** of the patterns
`!(pattern ...)`	Match **anything except** the given patterns
`*(pattern ...)`	Match **zero or more** of the given patterns (So, a null string, then the given patterns)
`+(pattern ...)`	Match **one or more** of the given patterns
`?(pattern ...)`	Match **zero or one** of the patterns

*(alice|hatter|hare) would match zero or more

occurrences of alice, hatter, and hare. So it

would match the null string, alice, alicehatter,

etc.

• +(alice|hatter|hare) would do the same except

not match the null string.

• ?(alice|hatter|hare) would only match the null

string, alice, hatter, or hare.

• @(alice|hatter|hare) would only match alice,

hatter, or hare.

• !(alice|hatter|hare) matches everything except alice, hatter, and hare

In the given example below, 1st case, for *, all the file contents were printed, including the null string(empty), 2nd case, +, all except the null string is considered, third being ?, showing null string or one of the 3 files, @ being exactly one of them and ! being any other files on the same directory as the files in question.

```
gayat@GSP-HP:~/SummerInternship$ cat *(123.txt|5|5678.txt)
this is updating the content to I can see if redirecting input and output is woring

sorry, working2 * 3  is a valid inequality.
gayat@GSP-HP:~/SummerInternship$ cat +(123.txt|5|5678.txt)
this is updating the content to I can see if redirecting input and output is woring

sorry, working2 * 3  is a valid inequality.
gayat@GSP-HP:~/SummerInternship$ cat ?(123.txt|5|5678.txt)
this is updating the content to I can see if redirecting input and output is woring

sorry, working2 * 3  is a valid inequality.
gayat@GSP-HP:~/SummerInternship$ cat @(123.txt|5|5678.txt)
this is updating the content to I can see if redirecting input and output is woring

sorry, working2 * 3  is a valid inequality.
gayat@GSP-HP:~/SummerInternship$ cat !(123.txt|5|5678.txt)
Alice, after her long sleep, decided to wake/ describe the characters of her dreams.
The Caterpillar and Alice looked at each other for some in silence: at last Caterpillar took t
he hookah out of its and addressed her in a languid, sleepy voice
hi hello testing
cat: test2705: Is a directory
gayat@GSP-HP:~/SummerInternship$
```

COMMAND SUBSTITUTION

Allows the user to use the standard output of a command to be assigned to a variable.

$(UNIX command)

So the command is run, and the output is assigned as the value of the expression

```
gayat@GSP-HP:~/SummerInternship$ echo $(pwd)
/home/gayat/SummerInternship
gayat@GSP-HP:~/SummerInternship$ echo $(ls $HOME)
Hello SummerInternship greet greet_user ().sh musician.sh ppt.sh script.sh snap test123
gayat@GSP-HP:~/SummerInternship$ echo $(ls $(pwd))
123.txt 5 5678.txt Untitled - Copy.txt:Zone.Identifier and hello.txt mouth, myfilecreated.txt
 new.txt test2705 time
gayat@GSP-HP:~/SummerInternship$ echo $(<123.txt)
this is updating the content to I can see if redirecting input and output is woring sorry, wo
rking
gayat@GSP-HP:~/SummerInternship$
```

Command substitution uses double quotes like variables and tilde expansion.

"When in doubt, use single quotes, unless the string contains variables or command substitutions, in which case use double quotes."

Cut command-→ a data filtering command that cuts in columns when you give the number of columns to be cut.

albums | musician.sh

home > gayat > SummerInternship > albums

```
1  Depeche Mode|Speak and Spell|Mute Records|1981
2  Depeche Mode|Some Great Reward|Mute Records|1984
3  Depeche Mode|101|Mute Records|1989
4  Depeche Mode|Violator|Mute Records|1990
5  Depeche Mode|Songs of Faith and Devotion|Mute Records|1991
```

```
gayat@GSP-HP:~/SummerInternship$ cut -f4 -d\| albums
1981
1984
1989
1990
1991
gayat@GSP-HP:~/SummerInternship$
```

(-d\| is to specify the character used as field delimiter) f4→ 4th field or column

Who command→ to see who all have logged in

```
gayat@GSP-HP:~/SummerInternship$ who
gayat    pts/1        2025-06-06 06:42
gayat@GSP-HP:~/SummerInternship$
```

TO just get the usernames. 1st case, the space itself is the delimiter since the spaceis what that separates the contents.

```
gayat@GSP-HP:~/SummerInternship$ who | cut -d' ' -f1
gayat
gayat@GSP-HP:~/SummerInternship$ who | cut -d\|  -f1
gayat    pts/1        2025-06-06 06:42
gayat@GSP-HP:~/SummerInternship$
```

PUSHD and POPD

Putting something onto a stack→ pushing→ pushd saves the current directory and switch

pushd [directory]

Taking something off the to→ popping→ popd removes the top of the stack and go back

popd

```
gayat@GSP-HP:~$ pwd
/home/gayat
gayat@GSP-HP:~$ pushd /etc
/etc ~
gayat@GSP-HP:/etc$ pwd
/etc
gayat@GSP-HP:/etc$ pushd /var
/var /etc ~
gayat@GSP-HP:/var$ popd
/etc ~
gayat@GSP-HP:/etc$ popd
~
gayat@GSP-HP:~$ pwd
/home/gayat
gayat@GSP-HP:~$
```

Command	**Stack contents**	**Result directory**
pushd lizard	*/home/you/lizard* */home/you*	*/home/you/lizard*
pushd /etc	*/etc* */home/you/lizard* */home/you*	*/etc*
popd	*/home/you/lizard* */home/you*	*/home/you/lizard*
popd	*/home/you*	*/home/you*
popd	<empty>	(error)