Chapter 3: Customising your environment

Environment- A collection of concepts that express the things a system with a set of tools does comfortably and understandably. (A collection of settings, tools, and configurations that define the context in which a system operates and interacts with users. It refers to the surroundings—physical or virtual—where tasks are performed. In computing, a virtual environment typically consists of software and tools designed to function cohesively, enabling users to work more efficiently and effectively. Like your working desk at an office where stationery is the tool set, or mobile phone screen where each icon and application is a tool.)

The environment can be customised according to the user's convenience (the same way we arrange the pens and books on a table or customise your phone screen with positions of the icons and widgets).

In general, the more customisation you have done, the more tailored to your personal needs—and therefore the more productive—your environment is.

So, UNIX shell also gives you files, directories, and tools to manipulate them as input and output. The customisation is based on how your input and output devices are, plus how and where you set the directories and files, their names, etc.

There are 4 most important and basic features that bash gives to customise our working environment.

1. Special files- Files read when you log into the bash, work in bash and when you log out of bash.

2. Aliases- Shortcuts for commands- You can give customisable keys or commands for complex commands to run them easily and effectively.

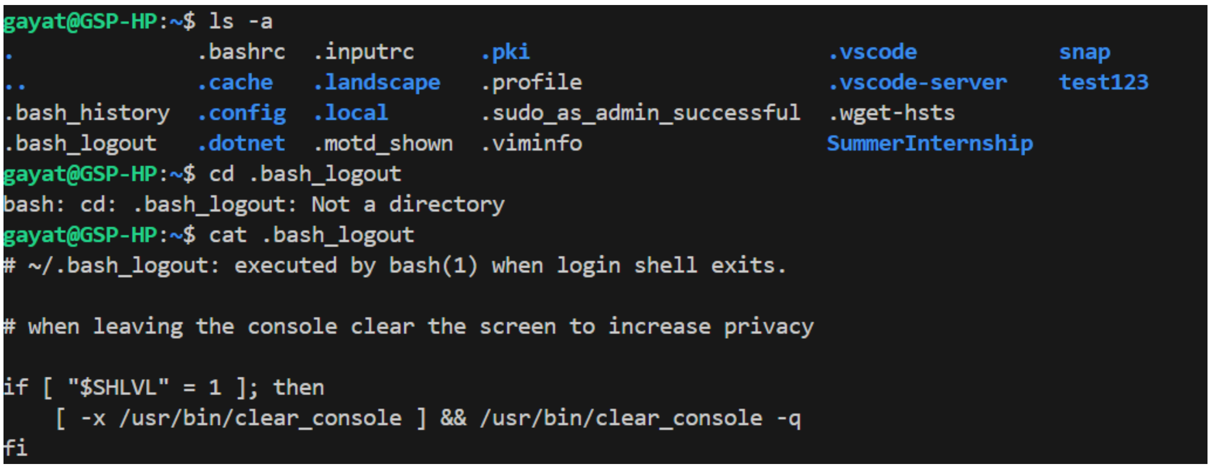
3. Options- Control for various aspects of your environment that can be turned off and on.

4. Variables- Changeable values referred to by a name. This can be customisable. Basically, you assign something with a meaning, so whenever you use that, you get the assigned meaning as output

So, as analogues, **Special files** are like your morning routine

File Name	What It Does	Analogy
.bash_profile- (Most important bash file)	Is read and run when you **log in** to a shell session- sets the environment when you start a shell	Morning routine at your desk- arranging the desk, setting your to-do list for the day.
.bashrc	Runs when you start a **new shell window**	Tools you always keep handy- like a new page of a notepad or a new pen. So, a new project/shell means a new set of notepads and pens.
.bash_logout	Runs when you **log out**	End-of-day clean-up routine- clearing the desk, switching the device off, arranging the tools back to the places.

For me, my PC doesn't have .bash_profile, so it uses .profile to set up the environment generally. Generally, .bash_profile has 2 synonyms- .bash_login from C shell and .profile and .login from Bourne shell and Korn shell's file called .profile.

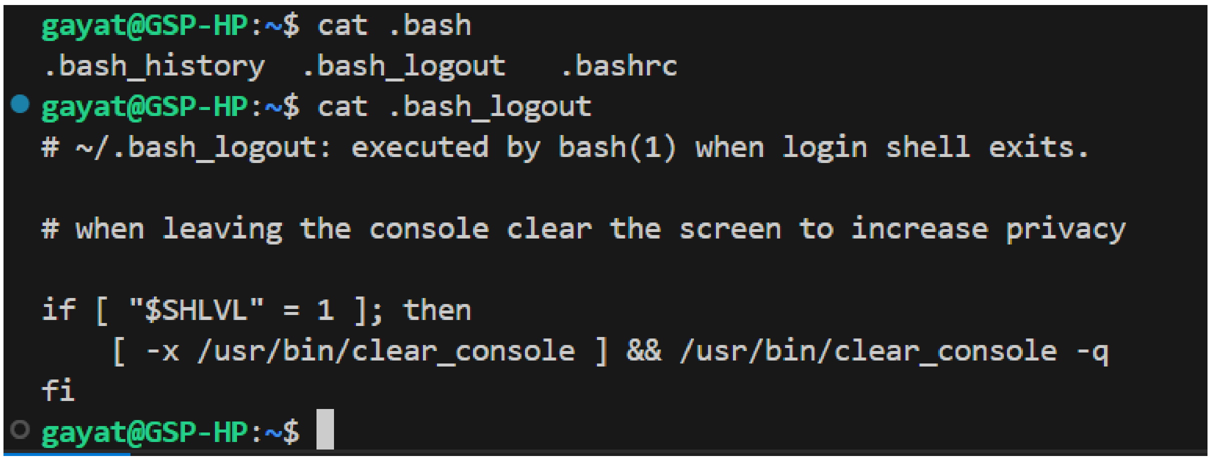


```
gayat@GSP-HP:~$ ls -a
.               .bashrc  .inputrc     .pki                        .vscode         snap
..              .cache   .landscape   .profile                    .vscode-server  test123
.bash_history   .config  .local       .sudo_as_admin_successful   .wget-hsts
.bash_logout    .dotnet  .motd_shown  .viminfo                    SummerInternship
gayat@GSP-HP:~$ cd .bash_logout
bash: cd: .bash_logout: Not a directory
gayat@GSP-HP:~$ cat .bash_logout
# ~/.bash_logout: executed by bash(1) when login shell exits.

# when leaving the console clear the screen to increase privacy

if [ "$SHLVL" = 1 ]; then
    [ -x /usr/bin/clear_console ] && /usr/bin/clear_console -q
fi
```

The best advantage is the fact that you can retain the **.profile f**ile while creating a new **.bash_profile**.

*When you start a new bash by typing bash, **.bashrc** is read by the system to start a new shell. This is required to start a new subshell.*

For logging out of the shell, the special file read is **.bash_logout, which is read to execute those commands to log out of the shell.**



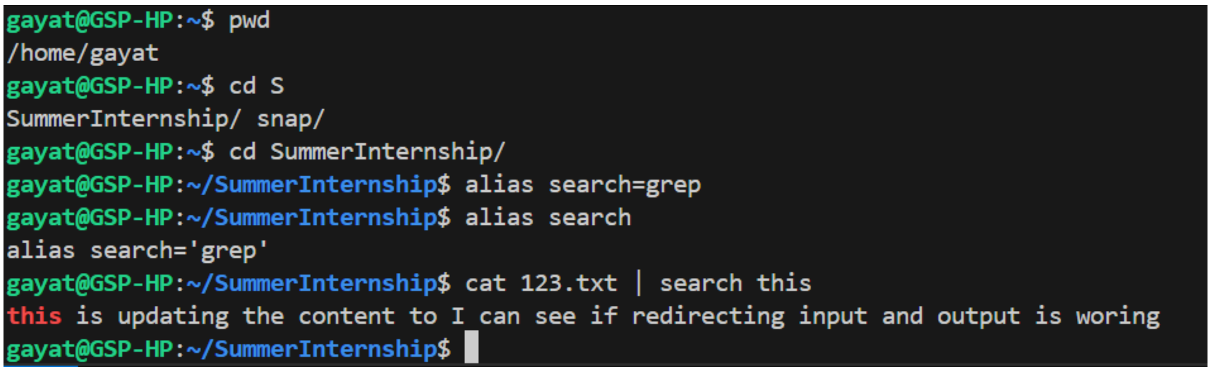
```
gayat@GSP-HP:~$ cat .bash
.bash_history  .bash_logout   .bashrc
gayat@GSP-HP:~$ cat .bash_logout
# ~/.bash_logout: executed by bash(1) when login shell exits.

# when leaving the console clear the screen to increase privacy

if [ "$SHLVL" = 1 ]; then
    [ -x /usr/bin/clear_console ] && /usr/bin/clear_console -q
fi
gayat@GSP-HP:~$
```

Aliases are like shortcuts and nicknames for tools that you work with.

Feature	What It Does	Analogy
Alias	Defines a shortcut for a longer command	Labelling your "stapler" instead of saying "Stapler Model 378XZ" every time

alias ll='ls -la --color=auto'..........................here, we are putting a shortcut for that particular command.

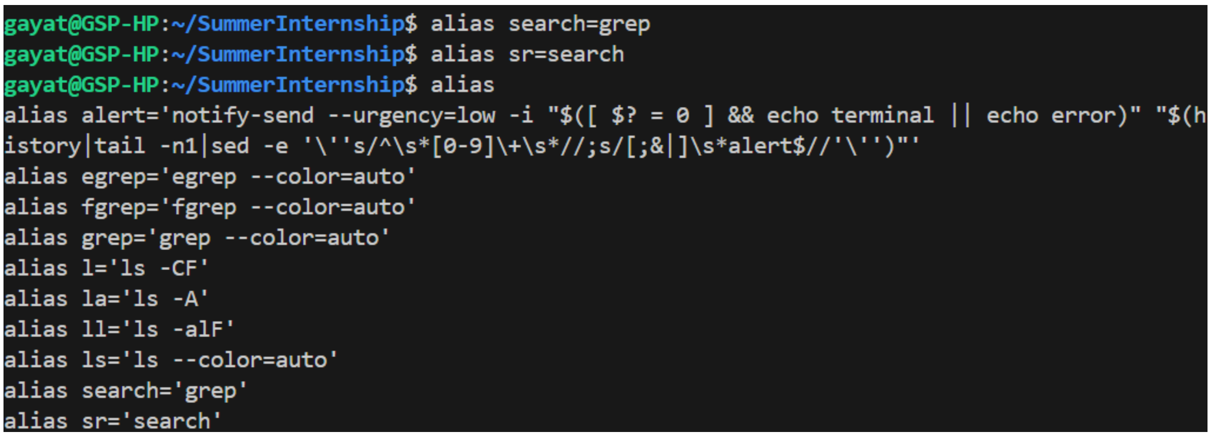
The format of running an alias→ **alias name=command,** with name being the shortcut for that particular command. So, whenever you use that name, you get that command run.



```
gayat@GSP-HP:~$ pwd
/home/gayat
gayat@GSP-HP:~$ cd S
SummerInternship/ snap/
gayat@GSP-HP:~$ cd SummerInternship/
gayat@GSP-HP:~/SummerInternship$ alias search=grep
gayat@GSP-HP:~/SummerInternship$ alias search
alias search='grep'
gayat@GSP-HP:~/SummerInternship$ cat 123.txt | search this
this is updating the content to I can see if redirecting input and output is woring
gayat@GSP-HP:~/SummerInternship$
```

Bash **replaces your alias with the full command** it refers to—**before** doing anything else. More like find and replace. Won't affect the special characters as alias is processed.

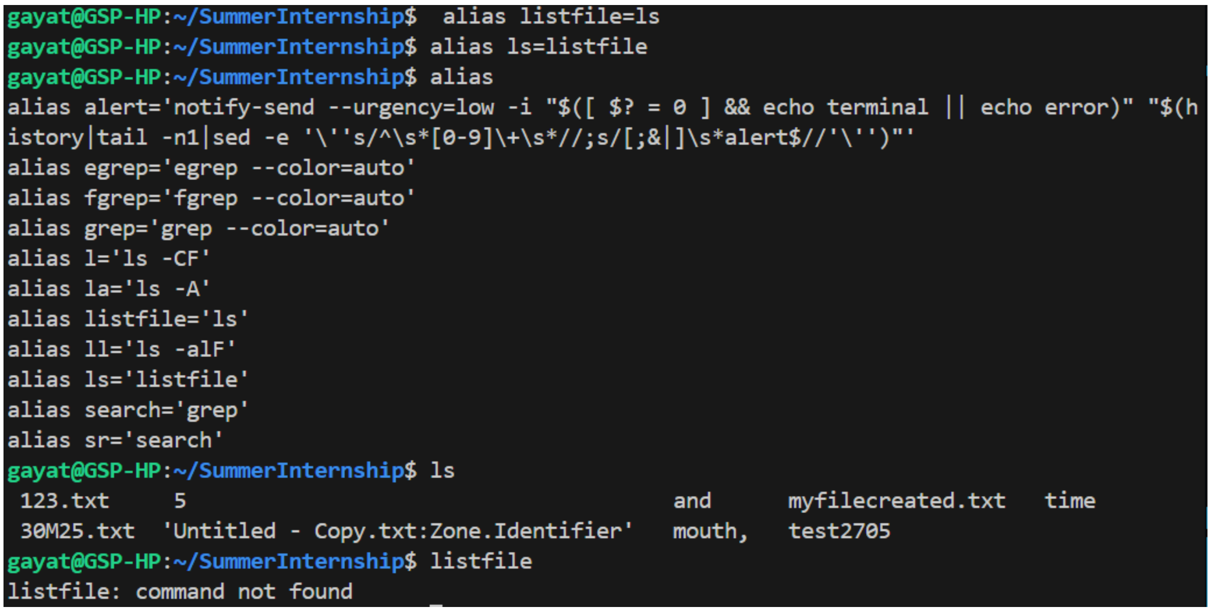
Also, you can alias an alias. So, search is given to grep, you alias search as sr.



```
gayat@GSP-HP:~/SummerInternship$ alias search=grep
gayat@GSP-HP:~/SummerInternship$ alias sr=search
gayat@GSP-HP:~/SummerInternship$ alias
alias alert='notify-send --urgency=low -i "$([ $? = 0 ] && echo terminal || echo error)" "$(h
istory|tail -n1|sed -e '\''s/^\s*[0-9]\+\s*//;s/[;&|]\s*alert$//'\'')"'
alias egrep='egrep --color=auto'
alias fgrep='fgrep --color=auto'
alias grep='grep --color=auto'
alias l='ls -CF'
alias la='ls -A'
alias ll='ls -alF'
alias ls='ls --color=auto'
alias search='grep'
alias sr='search'
```

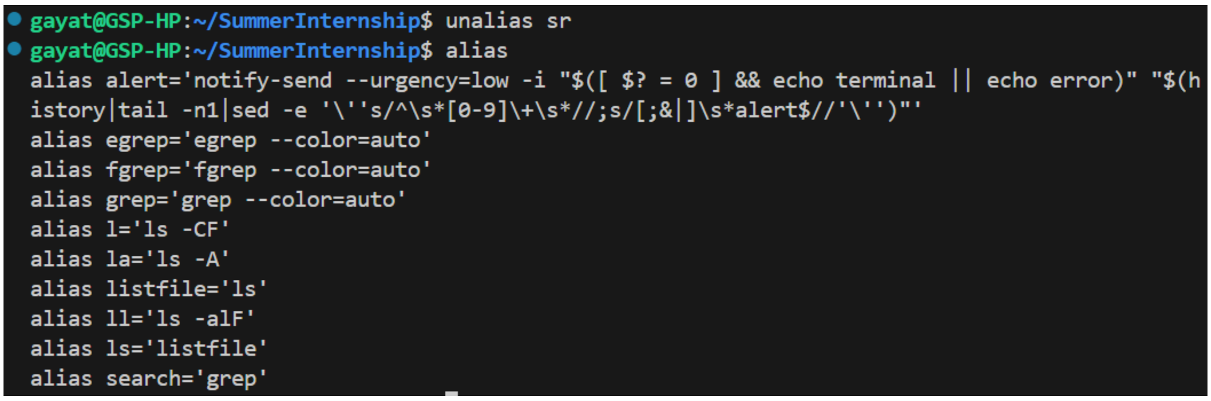
Bash prevents recursive aliasing where the alias is aliased using the command it is aliasing! Funny, isn't it?

So, in the below example, the bash prevented infinite recursion by not expanding an alias more than once. So, it stopped expansion after ls. No expansion for listfile.



```
gayat@GSP-HP:~/SummerInternship$  alias listfile=ls
gayat@GSP-HP:~/SummerInternship$ alias ls=listfile
gayat@GSP-HP:~/SummerInternship$ alias
alias alert='notify-send --urgency=low -i "$([ $? = 0 ] && echo terminal || echo error)" "$(h
istory|tail -n1|sed -e '\''s/^\s*[0-9]\+\s*//;s/[;&|]\s*alert$//'\'')"'
alias egrep='egrep --color=auto'
alias fgrep='fgrep --color=auto'
alias grep='grep --color=auto'
alias l='ls -CF'
alias la='ls -A'
alias listfile='ls'
alias ll='ls -alF'
alias ls='listfile'
alias search='grep'
alias sr='search'
gayat@GSP-HP:~/SummerInternship$ ls
 123.txt     5                                  and      myfilecreated.txt   time
 30M25.txt  'Untitled - Copy.txt:Zone.Identifier'   mouth,   test2705
gayat@GSP-HP:~/SummerInternship$ listfile
listfile: command not found
```

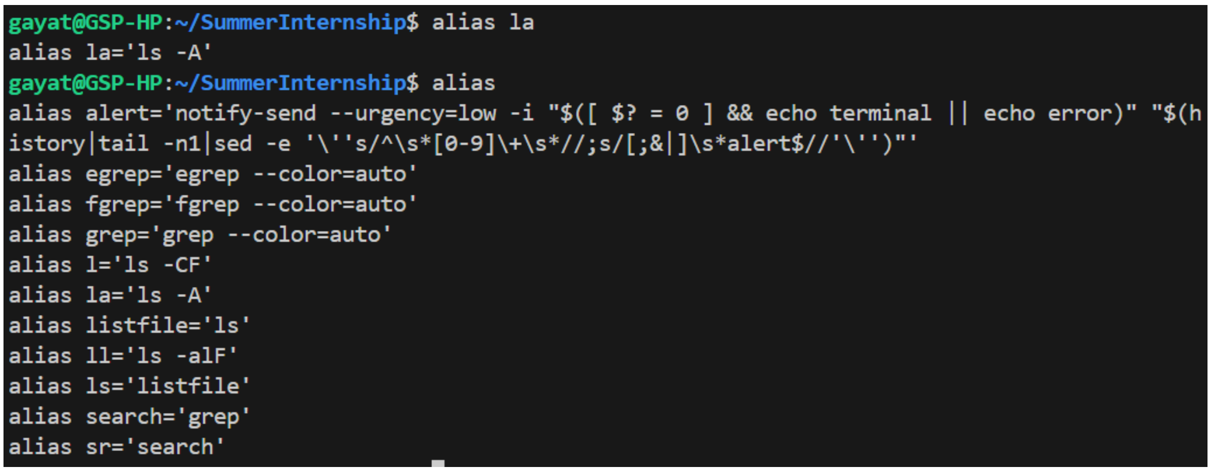
To remove the alias, use the ***unalias command name***



```
gayat@GSP-HP:~/SummerInternship$ unalias sr
gayat@GSP-HP:~/SummerInternship$ alias
alias alert='notify-send --urgency=low -i "$([ $? = 0 ] && echo terminal || echo error)" "$(h
istory|tail -n1|sed -e '\''s/^\s*[0-9]\+\s*//;s/[;&|]\s*alert$//'\'')"'
alias egrep='egrep --color=auto'
alias fgrep='fgrep --color=auto'
alias grep='grep --color=auto'
alias l='ls -CF'
alias la='ls -A'
alias listfile='ls'
alias ll='ls -alF'
alias ls='listfile'
alias search='grep'
```

Note: - The Alias command can do 3 things:

1. *Create an alias for a complex command*
2. *See what exactly a command or a control key does.*
3. *Just alias gives the list of all commands that have aliases.*



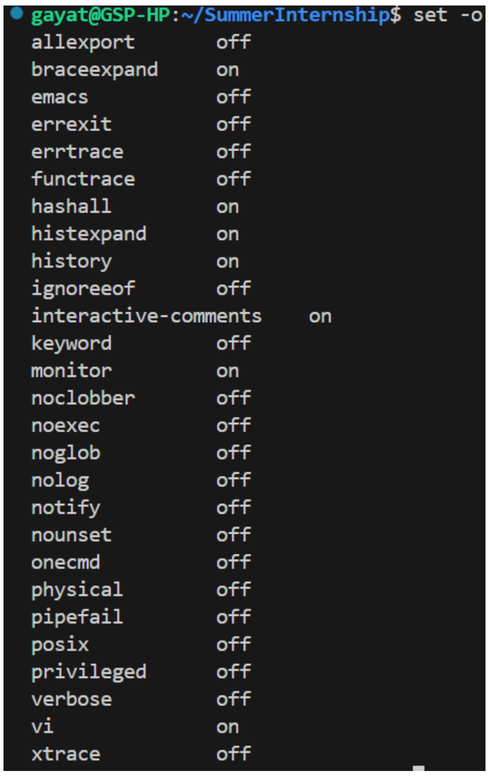
```
gayat@GSP-HP:~/SummerInternship$ alias la
alias la='ls -A'
gayat@GSP-HP:~/SummerInternship$ alias
alias alert='notify-send --urgency=low -i "$([ $? = 0 ] && echo terminal || echo error)" "$(h
istory|tail -n1|sed -e '\''s/^\s*[0-9]\+\s*//;s/[;&|]\s*alert$//'\'')"'
alias egrep='egrep --color=auto'
alias fgrep='fgrep --color=auto'
alias grep='grep --color=auto'
alias l='ls -CF'
alias la='ls -A'
alias listfile='ls'
alias ll='ls -alF'
alias ls='listfile'
alias search='grep'
alias sr='search'
```

Options are like customisable settings that can be turned on or off according to the user's preferences. You can set your editor mode to be vi or emacs, you can stop your shell from overwriting files. Using *set -o* to see all options available to turn on and off. Option is a method to change the shell's behaviour. So, a shell option is a setting that is either on or off.

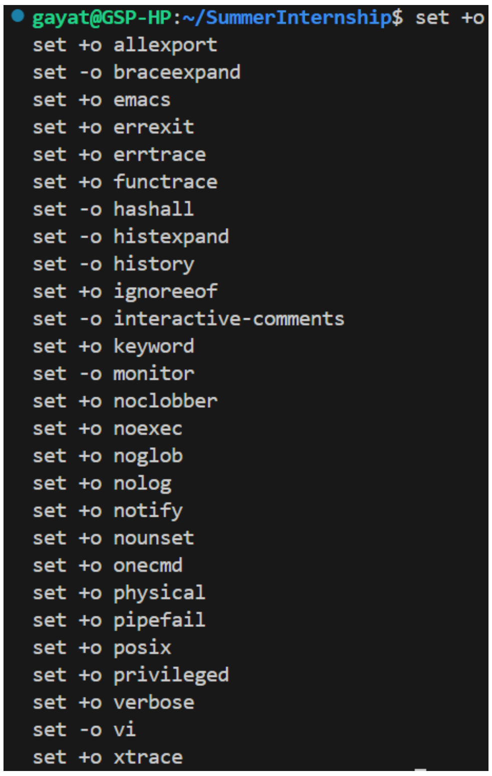
Command	**What It Controls**	**Analogy**
set -o noclobber	Prevents overwriting files accidentally	A safety lock on your shredder
set -o vi	Enables vi-style keyboard shortcuts	Switching your keyboard mode

Basic commands for options are set -o and set +o. They are used to control on and off of commands.

Command	**Meaning**
set -o OPTION	**Enable** a shell option
set +o OPTION	**Disable** a shell option



```
gayat@GSP-HP:~/SummerInternship$ set -o
allexport       off
braceexpand     on
emacs           off
errexit         off
errtrace        off
functrace       off
hashall         on
histexpand      on
history         on
ignoreeof       off
interactive-comments    on
keyword         off
monitor         on
noclobber       off
noexec          off
noglob          off
nolog           off
notify          off
nounset         off
onecmd          off
physical        off
pipefail        off
posix           off
privileged      off
verbose         off
vi              on
xtrace          off
```



```
gayat@GSP-HP:~/SummerInternship$ set +o
set +o allexport
set -o braceexpand
set +o emacs
set +o errexit
set +o errtrace
set +o functrace
set -o hashall
set -o histexpand
set -o history
set +o ignoreeof
set -o interactive-comments
set +o keyword
set -o monitor
set +o noclobber
set +o noexec
set +o noglob
set +o nolog
set +o notify
set +o nounset
set +o onecmd
set +o physical
set +o pipefail
set +o posix
set +o privileged
set +o verbose
set -o vi
set +o xtrace
```

So, set -o is used to show all the shell options in a human-readable form, marking on and off.

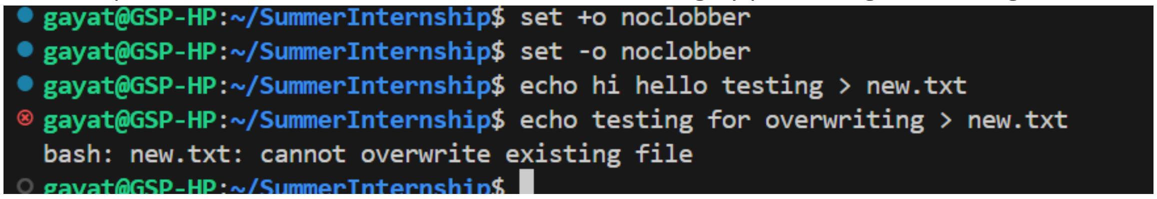
For set +o, it is used to show the same output, but in a format understood and reused by scripts, good to restore or save the shell state.

Note: ***Script-friendly output means***:

Output that is already formatted as valid shell commands, so it can be reused directly in scripts or with source. You can copy directly from the output and rerun it in a terminal. This is what is seen in set +o command by options.

Option	**Turn it ON**	**What it does**
emacs	*set -o emacs*	Enables Emacs-style command-line editing (default).
ignoreeof	*set -o ignoreeof*	Prevents logout with CTRL. Use exit instead.
noclobber	*set -o noclobber*	Prevents overwriting files with >. Use >! or `>
noglob	*set -o noglob*	Disables wildcard expansion (*, ?).
nounset	*set -o nounset*	Errors out if using unset variables.
vi	*set -o vi*	Enables vi editing mode for the command line.

For example, I tried set noclobber on and see if it's working by preventing overwriting on files.



```
gayat@GSP-HP:~/SummerInternship$ set +o noclobber
gayat@GSP-HP:~/SummerInternship$ set -o noclobber
gayat@GSP-HP:~/SummerInternship$ echo hi hello testing > new.txt
gayat@GSP-HP:~/SummerInternship$ echo testing for overwriting > new.txt
bash: new.txt: cannot overwrite existing file
gayat@GSP-HP:~/SummerInternship$
```

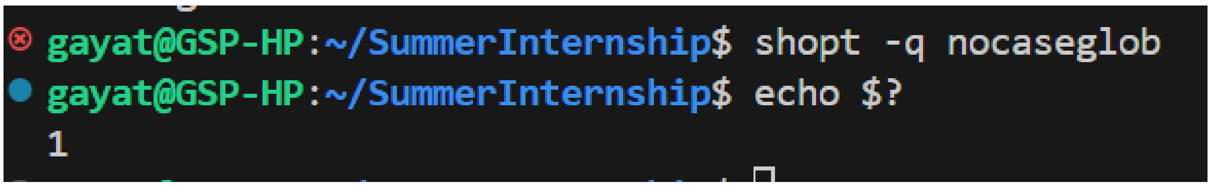
SHOPT: short for shell options

A new built-in command for configuring shell behaviour that isn't handled by set -o, as an optional replacement for options.

The format for this command is ***shopt option option-names***

Option	Meaning
-p	Displays a list of the settable options and their current values- shows if it's set or unset
-s	Sets each option name- shows the list of commands that are set/on
-u	Unset each option name- shows the list of commands that are unset/off
-q	Suppresses normal output; the return status indicates if a variable is set or unset
-o	Allows the values of the option names to be those defined for the -o option of the set command

How to see if the command is enabled or disabled.

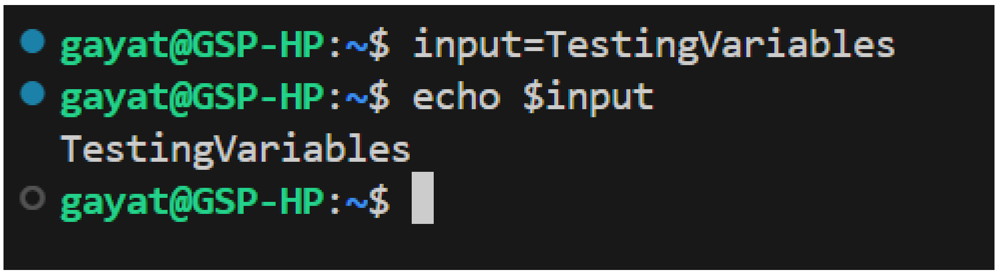


```
gayat@GSP-HP:~/SummerInternship$ shopt -q nocaseglob
gayat@GSP-HP:~/SummerInternship$ echo $?
1
```

Variables are like named containers, which contain information. So, they are characteristics that you want to customise but can't be turned on or off like in options. You assign that container to something, say a text or file. So, whenever you open that container, you get the contents assigned to it as the output. Or let's say it like using sticky notes, where you write something to do. So, the sticky note acts as a reminder. Like an alias, a shell variable is linked to a value or a meaning associated with it. By convention, the built-in variables have names with capital letters.

The format for creating a shell variable ***→ variable name=input/ value denoted.*** (No space between the = and words.)

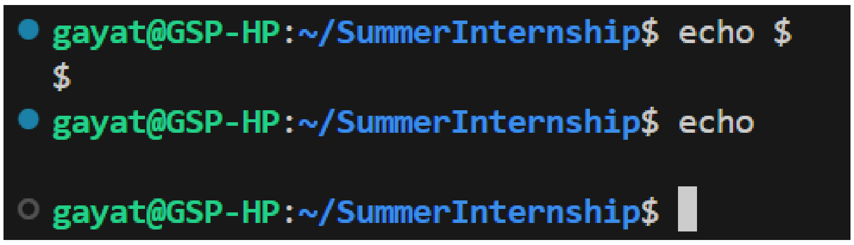
To use a variable, start the command with $, then the variable's name. Using echo command, you can print the value of a variable. So, *echo $variable name*



```
gayat@GSP-HP:~$ input=TestingVariables
gayat@GSP-HP:~$ echo $input
TestingVariables
gayat@GSP-HP:~$
```

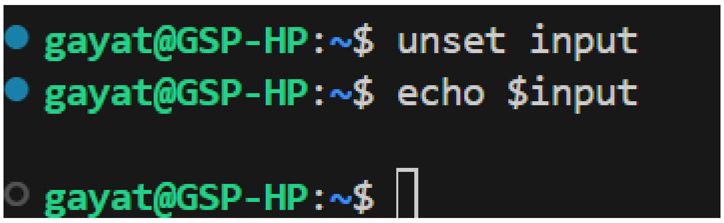
Here, the $ sign indicates that the word input is a variable. Note that variables are case-sensitive, so 'name' is not the same as 'Name'!

If the variable is undefined, the shell will print a blank line.



```
gayat@GSP-HP:~/SummerInternship$ echo $
$
gayat@GSP-HP:~/SummerInternship$ echo

gayat@GSP-HP:~/SummerInternship$
```

<u>You can unset a variable by typing unset with the variable</u>

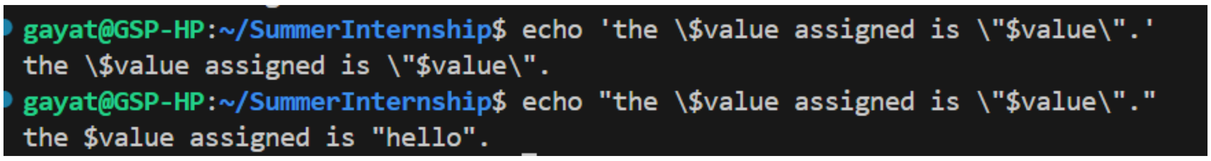


```
gayat@GSP-HP:~$ unset input
gayat@GSP-HP:~$ echo $input

gayat@GSP-HP:~$ 
```

<u>A special character that "survives" double quotes is the dollar sign, meaning that variables are evaluated.</u>

Note:

So, using \ and "", you can double quote $ and still it will work like usual. But, quoting with ' can remove the character.



```
gayat@GSP-HP:~/SummerInternship$ echo 'the \$value assigned is \"$value\".'
the \$value assigned is \"$value\".
gayat@GSP-HP:~/SummerInternship$ echo "the \$value assigned is \"$value\"."
the $value assigned is "hello".
```

Here, in the first command, I quoted using single quotes, causing it not to give the output as hello, as I named the shell variable hello. But using double quotes, I got the output. So use double quotes for variables.

<u>Built-in Variables</u>

Variable: Meaning

HISTCMD : The history number of the current command.

HISTCONTROL : A colon-separated list of patterns that control how commands are saved in history:

- ignorespace : Lines starting with a space are not saved.
- ignoredups : Lines that match the last command are not saved.
- erasedups : All previous matching lines are removed before saving the new one.
- ignoreboth : Enables both ignorespace and ignoredups.

HISTIGNORE : A colon-separated list of patterns that determine which command lines are skipped in history.

- Patterns must match the whole line (no implicit wildcards).
- An unescaped '&' matches the previous line. Use '\&' to match a literal ampersand.

HISTFILE : The file where the command history is saved.

Default: ~/.bash_history

HISTFILESIZE : The maximum number of lines to store in the history file.

Default: 500. The file is truncated to this size if necessary.

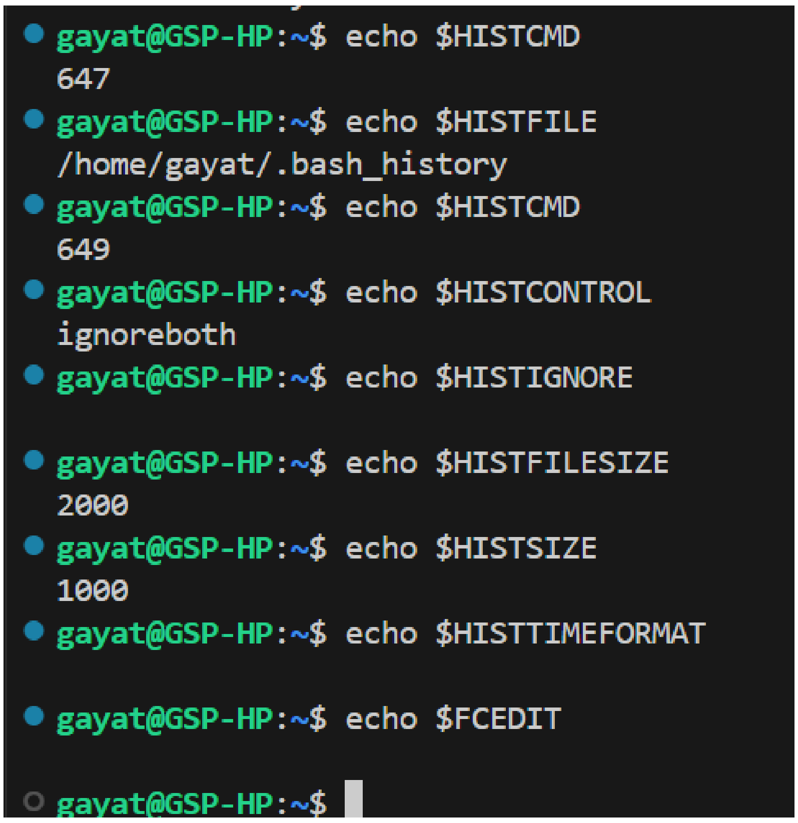
HISTSIZE : The maximum number of commands remembered in the current session.

Default: 500.

HISTTIMEFORMAT: If set, defines the strftime(3) format string used to show timestamps in `history` output.

Timestamps are preserved across shell sessions.

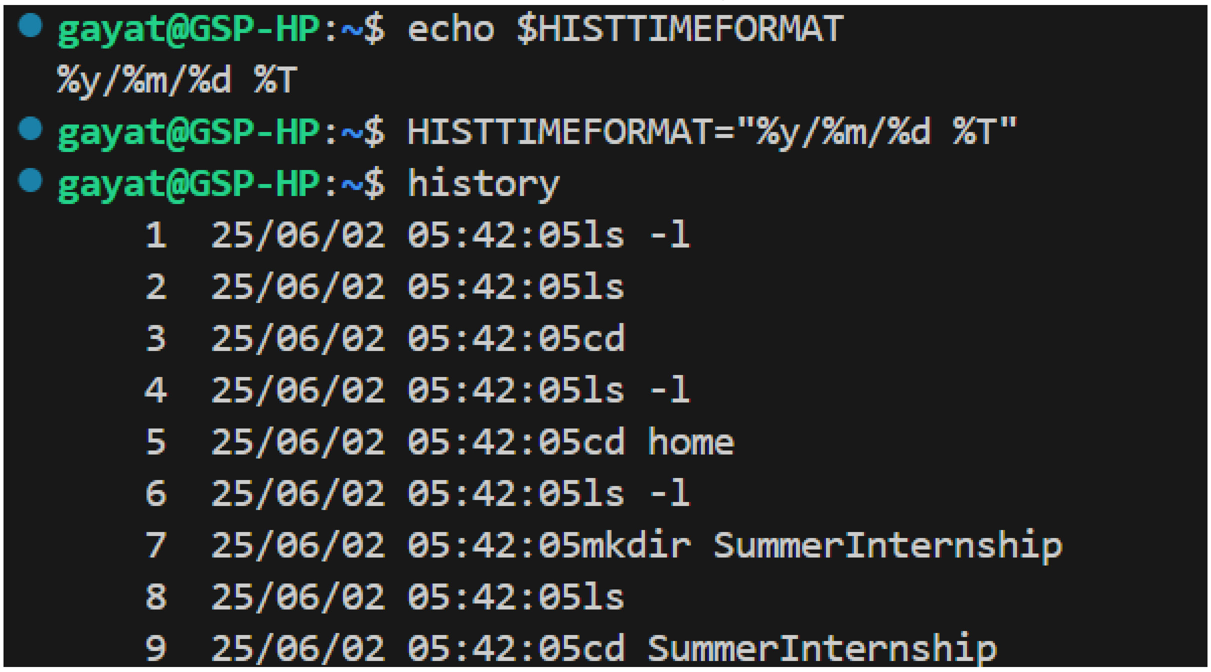
FCEDIT : Path to the editor used by the `fc` (fix command) built-in.



```
gayat@GSP-HP:~$ echo $HISTCMD
647
gayat@GSP-HP:~$ echo $HISTFILE
/home/gayat/.bash_history
gayat@GSP-HP:~$ echo $HISTCMD
649
gayat@GSP-HP:~$ echo $HISTCONTROL
ignoreboth
gayat@GSP-HP:~$ echo $HISTIGNORE

gayat@GSP-HP:~$ echo $HISTFILESIZE
2000
gayat@GSP-HP:~$ echo $HISTSIZE
1000
gayat@GSP-HP:~$ echo $HISTTIMEFORMAT

gayat@GSP-HP:~$ echo $FCEDIT

gayat@GSP-HP:~$
```

- **Using echo $HISTCMD, you get to see how many commands you ran, including echo $HISTCMD. So, in the end, you get to see your current command number.**
- **In my PC, it's currently ignoreboth for $HISTCONTROL, so it doesn't save statements starting with a space and ignores repeated commands.**
- **$HISTFILE gives the location for history.**
- **$HISTSIZE- Shows how many lines it can remember at a time**
- **$HISTFILESIZE- shows how many it can save at max**

- **$HISTIGNORE is used to see if a pattern is there and if so, don’t save it into the history.**
- **Here, I haven’t set $HISTTIMEFORMAT and $FCEDIR, so they are shown null when run.**



```
● gayat@GSP-HP:~$ echo $HISTTIMEFORMAT
  %y/%m/%d %T
● gayat@GSP-HP:~$ HISTTIMEFORMAT="%y/%m/%d %T"
● gayat@GSP-HP:~$ history
      1  25/06/02 05:42:05ls -l
      2  25/06/02 05:42:05ls
      3  25/06/02 05:42:05cd
      4  25/06/02 05:42:05ls -l
      5  25/06/02 05:42:05cd home
      6  25/06/02 05:42:05ls -l
      7  25/06/02 05:42:05mkdir SummerInternship
      8  25/06/02 05:42:05ls
      9  25/06/02 05:42:05cd SummerInternship
```

(skipping Mail variables)

Prompting Variables: Control how our command prompt looks and behaves like.

Variable	**Purpose**	**Used When**	**Example Value**
PS1	Main command prompt	Every command line	\u@\h:\w\$
PS2	Continuation prompt	Incomplete commands	>
PS3	Menu prompt for select	In scripts using select	Enter your choice:
PS4	Debug prefix (set -x)	Debugging shell scripts	+

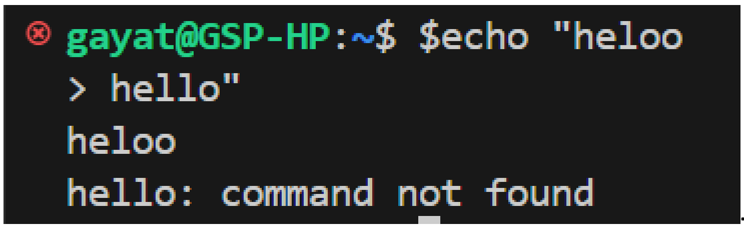
In case of PS1- primary prompt strings, common sequences followed are below:

Sequence	**Meaning**	**Example Output**
\u	Username	gayat
\h	Hostname (short)	GSP-HP
\H	Hostname (full)	GSP-HP.local
\w	Current working directory	~/SummerInternship
\W	Basename of current directory	SummerInternship

Sequence	**Meaning**	**Example Output**
\d	Date (e.g. Mon Jun 2)	Mon Jun 2
\t	Time (HH:MM:SS)	14:33:17
\T	Time (12-hour format)	02:33:17
\@	Time (12-hour AM/PM)	02:33 PM
\A	Time (HH:MM)	14:33
\!	History number of command	185
\#	Command number of session	24
\$	Shows # if root, $ if user	$
\n	Newline	

gayat@GSP-HP:~$ pwd

/home/gayat → The highlighted part is generated by PS1



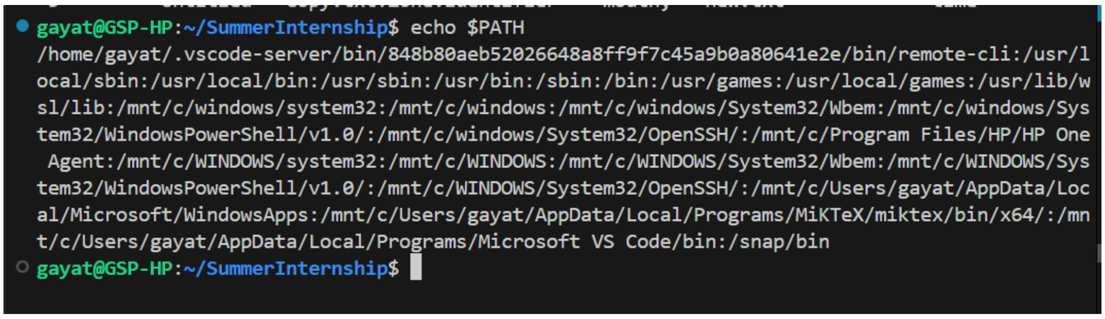
```
gayat@GSP-HP:~$ $echo "heloo
> hello"
heloo
hello: command not found
```

------Here the > sign came due to the incomplete command, and this is generated by PS2

Command Search Path

Path- A variable used to command the shell to find the commands you entered. (PATH is a variable that tells the shell (like bash) which directories to search for the *executables* (programs or scripts) that implement commands.)→ It's like drawers on a cupboard. It just gives you all the list of directories or locations of these commands' executable files, which the bash uses to run a command.

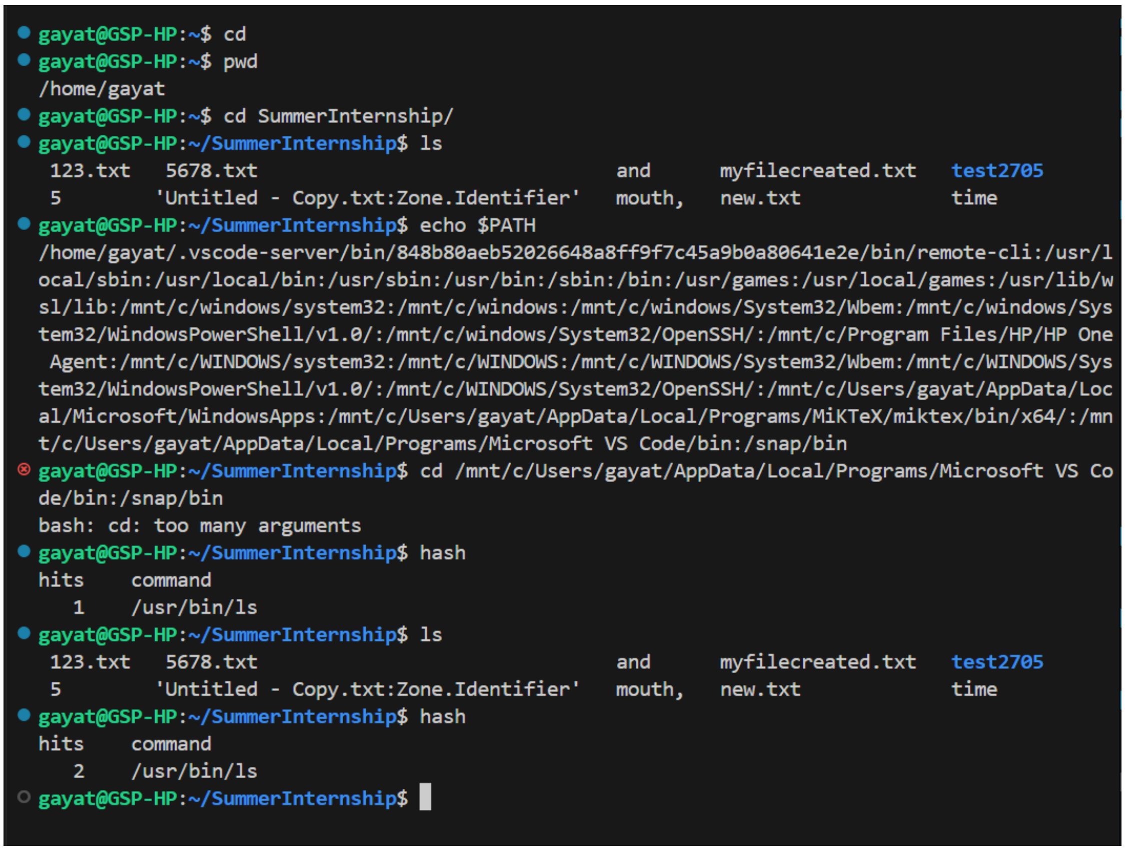
It stores a **list of directories (pathnames)** where the shell should look **for the executable files** that match the command you typed.



```
gayat@GSP-HP:~/SummerInternship$ echo $PATH
/home/gayat/.vscode-server/bin/848b80aeb52026648a8ff9f7c45a9b0a80641e2e/bin/remote-cli:/usr/l
ocal/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/usr/lib/w
sl/lib:/mnt/c/windows/system32:/mnt/c/windows:/mnt/c/windows/System32/Wbem:/mnt/c/windows/Sys
tem32/WindowsPowerShell/v1.0/:/mnt/c/windows/System32/OpenSSH/:/mnt/c/Program Files/HP/HP One
 Agent:/mnt/c/WINDOWS/system32:/mnt/c/WINDOWS:/mnt/c/WINDOWS/System32/Wbem:/mnt/c/WINDOWS/Sys
tem32/WindowsPowerShell/v1.0/:/mnt/c/WINDOWS/System32/OpenSSH/:/mnt/c/Users/gayat/AppData/Loc
al/Microsoft/WindowsApps:/mnt/c/Users/gayat/AppData/Local/Programs/MiKTeX/miktex/bin/x64/:/mn
t/c/Users/gayat/AppData/Local/Programs/Microsoft VS Code/bin:/snap/bin
gayat@GSP-HP:~/SummerInternship$
```

Command hashing

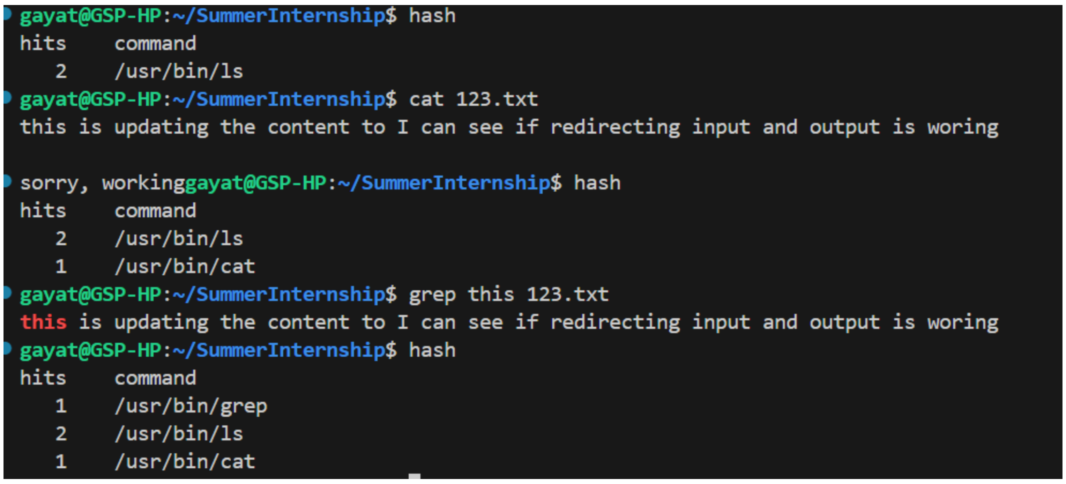
To speed up the process of searching in a long list of places for ***an external command***, Bash uses a **hash table**. It's Bash's internal memory of where it found the commands the first time. Instead of searching the PATH every time, it just checks the hash table. If the executable is changed to another file, bash will still go to the old place until it is cleared.



```
● gayat@GSP-HP:~$ cd
● gayat@GSP-HP:~$ pwd
  /home/gayat
● gayat@GSP-HP:~$ cd SummerInternship/
● gayat@GSP-HP:~/SummerInternship$ ls
   123.txt   5678.txt                          and      myfilecreated.txt   test2705
   5        'Untitled - Copy.txt:Zone.Identifier'   mouth,   new.txt             time
● gayat@GSP-HP:~/SummerInternship$ echo $PATH
  /home/gayat/.vscode-server/bin/848b80aeb52026648a8ff9f7c45a9b0a80641e2e/bin/remote-cli:/usr/l
  ocal/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/usr/lib/w
  sl/lib:/mnt/c/windows/system32:/mnt/c/windows:/mnt/c/windows/System32/Wbem:/mnt/c/windows/Sys
  tem32/WindowsPowerShell/v1.0/:/mnt/c/windows/System32/OpenSSH/:/mnt/c/Program Files/HP/HP One
   Agent:/mnt/c/WINDOWS/system32:/mnt/c/WINDOWS:/mnt/c/WINDOWS/System32/Wbem:/mnt/c/WINDOWS/Sys
  tem32/WindowsPowerShell/v1.0/:/mnt/c/WINDOWS/System32/OpenSSH/:/mnt/c/Users/gayat/AppData/Loc
  al/Microsoft/WindowsApps:/mnt/c/Users/gayat/AppData/Local/Programs/MiKTeX/miktex/bin/x64/:/mn
  t/c/Users/gayat/AppData/Local/Programs/Microsoft VS Code/bin:/snap/bin
⊗ gayat@GSP-HP:~/SummerInternship$ cd /mnt/c/Users/gayat/AppData/Local/Programs/Microsoft VS Co
  de/bin:/snap/bin
  bash: cd: too many arguments
● gayat@GSP-HP:~/SummerInternship$ hash
  hits    command
     1    /usr/bin/ls
● gayat@GSP-HP:~/SummerInternship$ ls
   123.txt   5678.txt                          and      myfilecreated.txt   test2705
   5        'Untitled - Copy.txt:Zone.Identifier'   mouth,   new.txt             time
● gayat@GSP-HP:~/SummerInternship$ hash
  hits    command
     2    /usr/bin/ls
○ gayat@GSP-HP:~/SummerInternship$ █
```

Here, on my screen, I have run ls once, so the hits for the command are 1. When I ran it again, it became 2.

Note that cd and pwd aren't listed on the hash table because the hash table is for external commands, which need executable files to run, while cd and pwd are built-in commands, hence, they don't exist as separate executable files in directories.



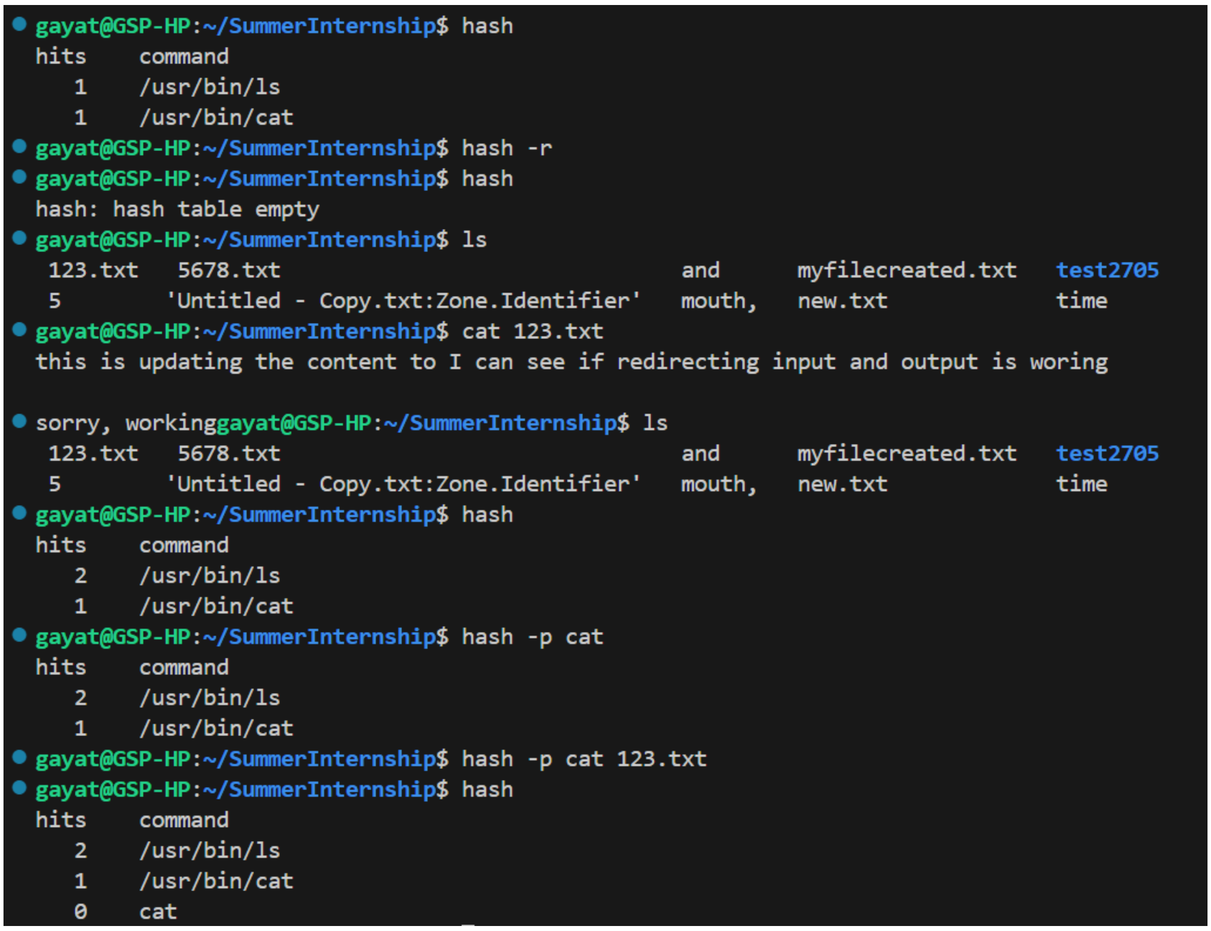
```
gayat@GSP-HP:~/SummerInternship$ hash
hits    command
   2    /usr/bin/ls
gayat@GSP-HP:~/SummerInternship$ cat 123.txt
this is updating the content to I can see if redirecting input and output is woring

sorry, workinggayat@GSP-HP:~/SummerInternship$ hash
hits    command
   2    /usr/bin/ls
   1    /usr/bin/cat
gayat@GSP-HP:~/SummerInternship$ grep this 123.txt
this is updating the content to I can see if redirecting input and output is woring
gayat@GSP-HP:~/SummerInternship$ hash
hits    command
   1    /usr/bin/grep
   2    /usr/bin/ls
   1    /usr/bin/cat
```

Hash- gives the list of all external commands run till then

Hash -r- clears the hash table

Hash -d commandname- removes that particular command from hash table

Hash -p commandname with filename- adds to the hash table without running the command.

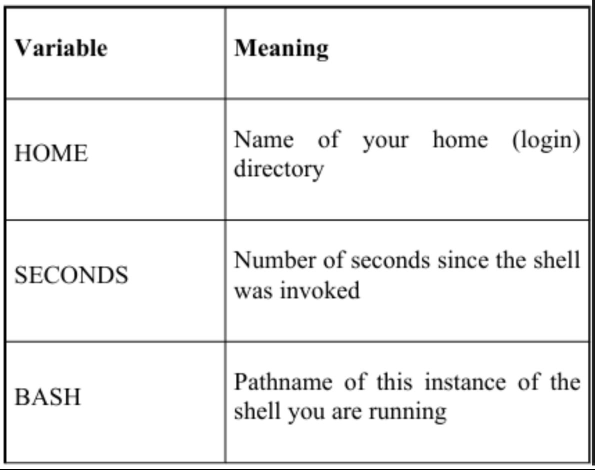


```
● gayat@GSP-HP:~/SummerInternship$ hash
  hits    command
     1    /usr/bin/ls
     1    /usr/bin/cat
● gayat@GSP-HP:~/SummerInternship$ hash -r
● gayat@GSP-HP:~/SummerInternship$ hash
  hash: hash table empty
● gayat@GSP-HP:~/SummerInternship$ ls
   123.txt   5678.txt                              and      myfilecreated.txt   test2705
   5         'Untitled - Copy.txt:Zone.Identifier'   mouth,   new.txt             time
● gayat@GSP-HP:~/SummerInternship$ cat 123.txt
  this is updating the content to I can see if redirecting input and output is woring

● sorry, workinggayat@GSP-HP:~/SummerInternship$ ls
   123.txt   5678.txt                              and      myfilecreated.txt   test2705
   5         'Untitled - Copy.txt:Zone.Identifier'   mouth,   new.txt             time
● gayat@GSP-HP:~/SummerInternship$ hash
  hits    command
     2    /usr/bin/ls
     1    /usr/bin/cat
● gayat@GSP-HP:~/SummerInternship$ hash -p cat
  hits    command
     2    /usr/bin/ls
     1    /usr/bin/cat
● gayat@GSP-HP:~/SummerInternship$ hash -p cat 123.txt
● gayat@GSP-HP:~/SummerInternship$ hash
  hits    command
     2    /usr/bin/ls
     1    /usr/bin/cat
     0    cat
```

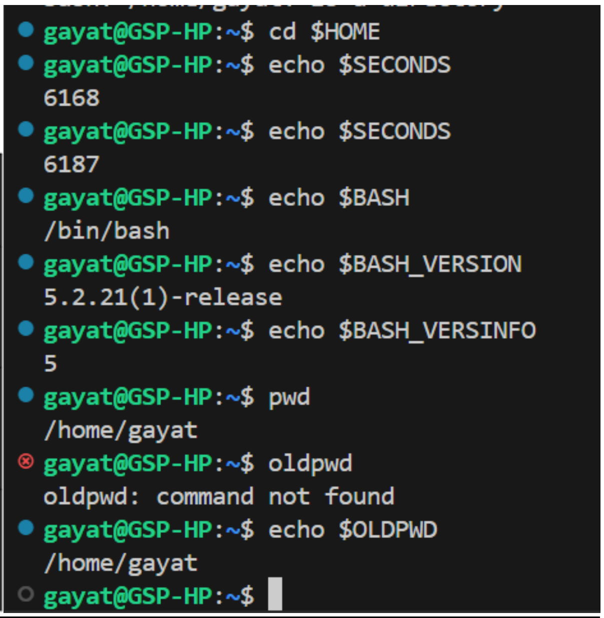
<u>**Directory search path and variables**</u>

CDPATH is an environment variable used to modify how the cd command works in the shell. Normally, cd looks for the directory in your current working directory. With CDPATH, cd will also search for the target directory in a list of specified directories, beyond just the current one. CDPATH is a colon (:) separated list of directories.

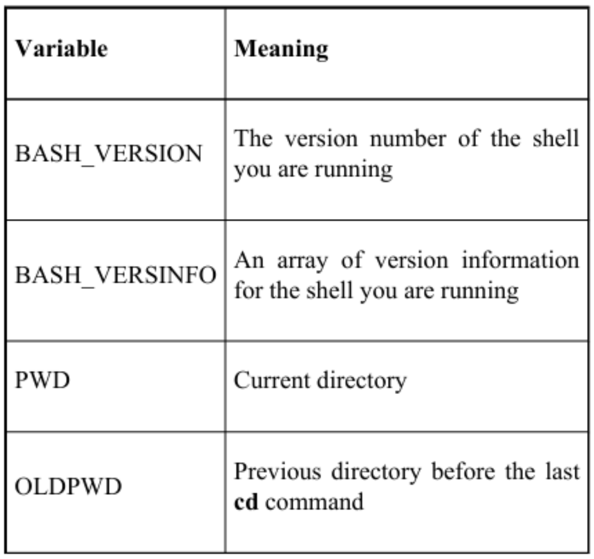
Miscellaneous Variables:



Variable	Meaning
HOME	Name of your home (login) directory
SECONDS	Number of seconds since the shell was invoked
BASH	Pathname of this instance of the shell you are running



```
● gayat@GSP-HP:~$ cd $HOME
● gayat@GSP-HP:~$ echo $SECONDS
  6168
● gayat@GSP-HP:~$ echo $SECONDS
  6187
● gayat@GSP-HP:~$ echo $BASH
  /bin/bash
● gayat@GSP-HP:~$ echo $BASH_VERSION
  5.2.21(1)-release
● gayat@GSP-HP:~$ echo $BASH_VERSINFO
  5
● gayat@GSP-HP:~$ pwd
  /home/gayat
⊗ gayat@GSP-HP:~$ oldpwd
  oldpwd: command not found
● gayat@GSP-HP:~$ echo $OLDPWD
  /home/gayat
○ gayat@GSP-HP:~$
```



Variable	Meaning
BASH_VERSION	The version number of the shell you are running
BASH_VERSINFO	An array of version information for the shell you are running
PWD	Current directory
OLDPWD	Previous directory before the last **cd** command

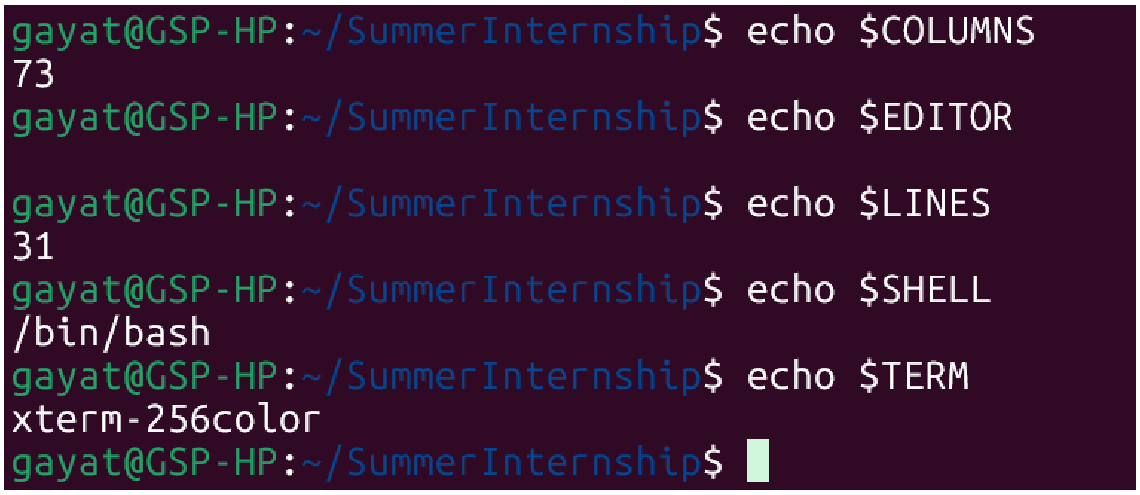
Environment Variables:

Environment variables are those used in a process to define the working environment for the user. When a subprocess is created, it inherits the environment variables of the parent process. Some of the built-in variables are environment variables- HOME, PATH, PWD, MAIL etc.

Environment variables are important. Most .bash_profile files include definitions of the environment variables.

Feature	**Environment Variables**	**Standard/Shell Variables**
Scope	Inherited by subprocesses	Only exist in current shell

Feature	**Environment Variables**	**Standard/Shell Variables**
Created with	export VAR=value	VAR=value
Visible to commands	Yes	No
Use case	Configure subprocess behavior	Temporary values in shell



```
gayat@GSP-HP:~/SummerInternship$ echo $COLUMNS
73
gayat@GSP-HP:~/SummerInternship$ echo $EDITOR

gayat@GSP-HP:~/SummerInternship$ echo $LINES
31
gayat@GSP-HP:~/SummerInternship$ echo $SHELL
/bin/bash
gayat@GSP-HP:~/SummerInternship$ echo $TERM
xterm-256color
gayat@GSP-HP:~/SummerInternship$
```

In the above terminal example, they are all standard variables.

COLUMNS AND LINES give the number of columns and lines on the current shell.

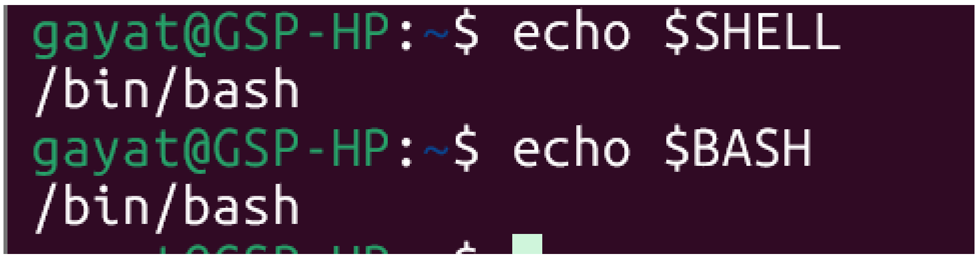
Standard variables can be the environment variables via the export command, or they are called the local variables for the current shell. So, not all shell variables are environment variables. Additionally, all environment variables in your shell are also standard shell variables.

TERM is an environment variable which is important for any programs that use the screen.

Other variables:

Both bash and shell give the same output, but serve slightly different purposes.

BASH is set to the pathname of the current shell, whether it is an interactive shell or not. SHELL, on the other hand, is set to the name of your login shell, which may be a completely different shell.



```
gayat@GSP-HP:~$ echo $SHELL
/bin/bash
gayat@GSP-HP:~$ echo $BASH
/bin/bash
```

<u>**The environment File**</u>:

The definitions to tell the shell which other variables, options and aliases other than environment variables are used are present in the environment file. Bash's default file is .bashrc, which is used when we start a new shell.

The idea evolved from the C shell's .cshrc file.

General rule to follow- a few definitions in .bash_profile and as many as possible in the environment file (here, .bashrc)

The important things necessary in .bash_profile are environment variables and their exports, commands to run when the user is logged out.