Navigating Linux System Commands

A guide for beginners to the Shell and GNU coreutils

Sayan Ghosh

July 26, 2024

IIT Madras
BS Data Science and Applications

Disclaimer
This document is a companion activity book for the System Commands (BSSE2001) course taught by **Prof. Gandham Phanikumar** at **IIT Madras BS Program.** This book contains resources, references, questions and solutions to some common questions on Linux commands, shell scripting, grep, sed, awk, and other system commands.

This was prepared with the help and guidance of the course instructors:

Santhana Krishnan and **Sushil Pachpinde**

Colophon
This document was typeset with the help of KOMA-Script and LaTeX using the kaobook class.
The source code of this book is available at:
`https://github.com/sayan01/se2001-book`
(You are welcome to contribute!)

Edition
Compiled on July 26, 2024

UNIX is basically a simple operating system, but you have to be a genius to understand the simplicity.

– Dennis Ritchie

Preface

Through this work I have tried to make learning and understanding the basics of Linux fun and easy. I have tried to make the book as practical as possible, with many examples and exercises. The structure of the book follows the structure of the course *BSSE2001 - System Commands*, taught by **Prof. Gandham Phanikumar** at **IIT Madras BS Program.** .

The book takes inspiration from the previous works done for the course,

- Sanjay Kumar's Github Repository
- Cherian George's Github Repository
- Prabuddh Mathur's TA Sessions

as well as external resources like:

- Robert Elder's Blogs and Videos
- Aalto University, Finland's Scientific Computing - Linux Shell Crash Course

The book covers basic commands, their motivation, use cases, and examples. The book also covers some advanced topics like shell scripting, regular expressions, and text processing using `sed` and `awk`.

This is not a substitute for the course, but a companion to it. The book is a work in progress and any contribution is welcome at `https://github.com/sayan01/se2001-book`

Sayan Ghosh

Contents

List of Figures

List of Tables

1 Shell Variables

We have seen how to execute commands in the linux shell, and also how to combine those commands to perform complex tasks. However, to make really powerful scripts, we need to store the output of commands, and also store intermediate results. This is where variables come in. In this chapter, we will learn how to create, manipulate, and use variables in the shell.

1.1 Creating Variables

There are two types of variables in the shell: **Environment Variables** and **Shell Variables**. Environment variables are accessible to all processes running in the environment, while shell variables are only accessible to the shell in which they are created.

Definition 1.1.1 (Environment Variables) An environment variable is a variable that is accessbile to all processes running in the environment. It is a key-value pair. It is created using the `export` command. It can be accessed using the $ followed by the name of the variable (e.g. $HOME) or using the `printenv` command. They are part of the environment in which a process runs. For example, a running process can query the value of the TEMP/TMPDIR environment variable to discover a suitable location to store temporary files, or the HOME or USER variable to find the directory structure owned by the user running the process.

Definition 1.1.2 (Shell Variables) A shell variable is a variable that is only accessible to the shell in which it is created. It is a key-value pair. It is created using the = operator. It can be accessed using the $ followed by the name of the variable (e.g. $var). They are local to the shell in which they are created.

Let us see how to create a shell variable.

```
$ var="Hello World"
```

This creates a shell variable `var` with value `Hello World`. If the value of our variable contains spaces, we need to enclose it in quotes. It is important to note that there should be **no spaces** around the = operator.

Variable names can contain letters, numbers, and underscores, but they cannot start with a number.

Similarly, for an environment variable, we use the `export` command.

```
$ export var="Hello World"
```

This creates an environment variable `var` with value `Hello World`. The difference between a shell variable and an environment variable is that the environment variable is accessible to all processes running in the environment, while the shell variable is only accessible to the shell in which it is created.

A variable can also be exported after it is created.

```
$ var="Hello World"
$ export var
```

1.2 Printing Variables to the Terminal

To access the value of a variable, we use the `\$` operator followed by the name of the variable. It is only used when we want to get the value of the variable, and not for setting the value.

```
$ var="Hello World"
$ echo $var
Hello World
```

However, it is often required to enclose the variable name in braces to avoid ambiguity when concatenating with other alpha-numeric characters.

Here we want to concatenate the values of the variables `date` and `time` with an underscore in between. However, the shell things that the first variable we are accessing is `date_`, and the second variable is `time`. This gives an empty first variable. To fix this ambiguity, we enclose the variable name in braces.

```
$ date="2024_07_30"
$ time="08:30:00"
$ echo "$date_$time"
08:30:00
$ echo "${date}_${time}"
2024_07_30_08:30:00
```

Remark 1.2.1 If we want to print the dollar symbol literally, we need to escape it using the backslash character, or surround it in single quotes, not double quotes.

```
$ echo '$USER is' "$USER"
$USER is sayan
$ echo "\$HOSTNAME is $HOSTNAME"
$HOSTNAME is rex
```

Here we are using the `echo` command to print the value of the variable `var` to the terminal.

1.2.1 Echo Command

The `echo` command displays a line of text on the terminal. It is commonly used in shell scripts to display a message or output of a command. It is also used to print the value of a variable.

On most shells **echo** is actually a built-in command, not an external program. This means that the shell has a built-in implementation of the echo command, which is faster than calling an external program.

Although the `echo` binary might be present on your system, it is not the one being called when executing `echo`. This is usually not an issue

for most use cases, however, you should be aware which `echo` you are running, so that you can refer to the correct documentation.

When using `man echo`, we get the documentation of `echo` binary distributed through GNU core utils, whereas when we using `help echo`, we get the documentation of the `echo` which is built-in into the bash shell.

```
$ man echo |  sed '/^$/d' | head -n15
ECHO(1)                          User Commands                          ECHO(1)
NAME
       echo - display a line of text
SYNOPSIS
       echo [SHORT-OPTION]... [STRING]...
       echo LONG-OPTION
DESCRIPTION
       Echo the STRING(s) to standard output.
       -n     do not output the trailing newline
       -e     enable interpretation of backslash escapes
       -E     disable interpretation of backslash escapes (default)
       --help display this help and exit
       --version
              output version information and exit
       If -e is in effect, the following sequences are recognized:
```

```
$ help echo | head -n20 | sed '/^ +$/d'
echo: echo [-neE] [arg ...]
    Write arguments to the standard output.
    Display the ARGs, separated by a single space character and followed by a
    newline, on the standard output.
    Options:
      -n    do not append a newline
      -e    enable interpretation of the following backslash escapes
      -E    explicitly suppress interpretation of backslash escapes
    'echo' interprets the following backslash-escaped characters:
      \a    alert (bell)
      \b    backspace
      \c    suppress further output
      \e    escape character
      \E    escape character
      \f    form feed
      \n    new line
      \r    carriage return
```

The options of both the `echo` commands look similar, however, GNU core-utils `echo` also has the support for two long options. These are options with two dashes, and are more descriptive than the short options. However, these are not present in the built-in `echo` command.

Thus, if we are reading the man page of `echo`, we might think that the long options will work with the default `echo`, but it will not.

```
$ type -a echo
echo is a shell builtin
echo is /sbin/echo
```

When we call the `echo` executable with its path, the executable gets executed instead of the built-in. This supports the long options. The built-in version simply prints the long option as text.

```
echo is /bin/echo
echo is /usr/bin/echo
echo is /usr/sbin/echo
```

```
$ echo --version
--version
```

```
$ /bin/echo --version
echo (GNU coreutils) 9.5
Copyright (C) 2024 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <https://gnu.org/
    licenses/gpl.html>.
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

Written by Brian Fox and Chet Ramey.
```

Escape Characters

The `echo` command also supports escape characters. These are special characters that are used to format the output of the `echo` command.

However, to use these escape characters, we need to use the `-e` option. The following list of escape characters are supported by the `echo` command is taken from the `help echo` output as-is.

```
'echo' interprets the following backslash-escaped characters:
      \a        alert (bell)
      \b        backspace
      \c        suppress further output
      \e        escape character
      \E        escape character
      \f        form feed
      \n        new line
      \r        carriage return
      \t        horizontal tab
      \v        vertical tab
      \\        backslash
      \0nnn     the character whose ASCII code is NNN (octal).
    NNN can be
                0 to 3 octal digits
      \xHH      the eight-bit character whose value is HH (
    hexadecimal).  HH
                can be one or two hex digits
      \uHHHH    the Unicode character whose value is the
    hexadecimal value HHHH.
                HHHH can be one to four hex digits.
      \UHHHHHHHH the Unicode character whose value is the
    hexadecimal value
                HHHHHHHH. HHHHHHHH can be one to eight hex digits.
```

The backspace character `\b` moves the cursor one character to the left. This is useful to overwrite a previously typed character.

```
$ echo -e "abc\bd"
abd
```

The suppress further output character `\c` suppresses the output of any text present after it.

```
$ echo -e "abc\cde"
abc
```

The escape character `\e` is used to escape the escape character after it, but continues printing after that.

The form feed character `\f` and vertical tab character `\v` moves the cursor to the next line, but does not move the cursor to the start of line, remaining where it was in the previous line.

```
$ echo -e "abc\ede"
abce
```

```
$ echo -e "abc\fde"
abc
   de
```

```
$ echo -e "abc\vde"
abc
   de
```

The carriage return character `\r` moves the cursor to the start of the line. This can be used to overwrite some parts of the previously written line. Characters not overwritten remain intact. Here the `de` overwrite the `ab` but the `c` remains intact.

```
$ echo -e "abc\rde"
dec
```

The new line character `\n` moves the cursor to the next line and the cursor to the start of the line. This is same as performing `\f\r`.

```
$ echo -e "abc\nde"
abc
de
```

The horizontal tab character `\t` moves the cursor to the next tab stop.

```
$ echo -e "abc\tde"
abc     de
```

The octal and hexadecimal characters can be used to print the character with the given ASCII code. Here, `\x41` is the ASCII code for `A` and `\0132` is the ASCII code for `Z`.

```
$ echo -e "\0132"
Z
$ echo -e "\x41"
A
```

1.2.2 Accessing and Updating Numeric Variables

If the variable is a number, we can perform arithmetic operations on it in a mathematical context. [1]

1: There are no data types in bash. A variable can store a number, a string, or any other data type. Every variable is treated as a string, unless inside a mathematical context.

Basic Arithmetic

Basic Arithmetic operations such as addition, subtraction, multiplication, division, and modulo can be performed using the `(( ))` construct.

Exponentiation can be performed using the `**` operator, similar to Python.

The ^ operator is reserved for bitwise XOR.

```
$ var=5
$ echo $var
5
$ echo $((var+5))
10
$ echo $((var-5))
0
$ echo $((var*5))
25
$ echo $((var/5))
1
$ echo $((var%5))
0
$ echo $((var**2))
25
```

Bitwise Operations

Bash also supports bitwise operations. The bitwise NOT operator is ~, the bitwise AND operator is &, the bitwise OR operator is |, and the bitwise XOR operator is ^. They operate on the binary representation of the number.

AND operation: The result is 1 if both bits are 1, otherwise 0.
OR operation: The result is 1 if either of the bits is 1, otherwise 0.
XOR operation: The result is 1 if the bits are different, otherwise 0.
NOT operation: The result is the complement of the number.

To understand why the result of the bitwise NOT operation is -6, we need to understand how the numbers are stored in the computer. The number 5 is stored as 00000101 in binary. The bitwise NOT operation inverts the bits, so 00000101 becomes 11111010. This is the two's complement representation of -6. Two's complement is how computers store negative numbers. This is better than one's complement (just flipping the bits of the positive number) as it gives a single value of zero, which is its own additive inverse. Read more about two's complement here.

```
$ echo $((var^2))
7
$ echo $((var&3))
1
$ echo $((var|3))
7
$ echo $((~var))
-6
```

Comparison Operations

The comparision operators return 1 if the condition is true, and 0 if the condition is false. This is the opposite of how the exit codes work in bash, where 0 means success and 1 means failure. However, we will see later that the mathematical enviroment actually exits with a zero exit code if the value is non-zero, fixing this issue.

```
$ echo $((var>5))
0
$ echo $((var>=5))
1
$ echo $((var<5))
0
$ echo $((var<=5))
1
$ echo $((var==5))
1
$ echo $((var!=5))
0
```

Increment and Decrement

Bash supports both pre-increment and post-increment, as well as pre-decrement and post-decrement operators. The difference between pre and post is that the pre operator increments the variable before using it, while the post operator increments the variable after using it.

Although `var++` increases the value, the updated value is not returned, thus the output of echo remains same as earlier. We can reprint the variable to confirm the change.

```
$ echo $((++var))
6
$ echo $((var++))
6
$ echo $var
7
$ echo $((--var))
```

```
6
$ echo $((var--))
6
$ echo $var
5
```

Assignment and Multiple Clauses

We can also perform multiple arithmetic operations in a single line. The result of the last operation is returned.

```
$ echo $((a=5, b=10, a+b))
15
$ echo $((a=5, b=10, a+b, a*b))
50
```

The evaluation of an assignment operation is the value of the right-hand side of the assignment. This behaviour helps in chaining multiple assignment operations.

```
$ echo $((a=5, b=10))
10
$ echo $((a=b=7, a*b))
49
```

Floating Point Arithmetic

Bash does not support floating point arithmetic.

```
$ echo $((10/3))
3
$ echo $((40/71))
0
```

However, we can use the `bc` command to perform floating point arithmetic. Other commands that support floating point arithmetic are `awk` and `perl`. We can also use the `printf` command to format the output of the floating point arithmetic after performing integer arithmetic in bash.

```
$ printf "%.2f\n" "$((10**4 * 10/3))e-4"
3.33
$ printf "%.2f%%\n" "$((10**4 * 40/71))e-4"
56.33%
```

```
$ awk 'BEGIN { printf "%.2f%%", (40/71*100) }'
56.34%
```

Working with different bases

Bash supports working with different bases. The bases can be in the range $[2, 64]$.

The syntax to refer to a number in a non-decimal base is:

```
base#number
```

We can use the `0x` prefix shortcut for hexadecimal numbers, and the `0` prefix shortcut for octal numbers.

We can also mix bases in the same expression.

31 in octal is same as 25 in decimal. This leads to the joke that programmers get confused between Haloween (31 Oct) and Christmas (25 Dec).

```
$ echo $((2#1000 + 2#101))
13
$ echo $((8#52 + 8#21))
59
$ echo $((052 + 021))
59
$ echo $((16#1a + 16#2a))
68
$ echo $((0x1a + 0x2a))
68
$ echo $((8#31 - 25))
0
```

If we only want to evaluate the expression and not print it, we can use the `(( ))` construct without the `echo` command.

```
$ var=5
$ ((var++))
$ echo $var
6
```

There are also other ways to perform arithmetic operations in bash, such as using the `expr` command, or using the `let` command.

The `expr` command is an external command, and not a shell built-in. Thus, it does not have access to the shell variables. If we want to use the values of the variables, we have to expand them first. The `*` operator is a glob that matches all files in the current directory. To prevent this, we need to escape the `*` operator.

```
$ var=5
$ expr $var + 5
10
$ expr $var \* 5
25
```

The `let` command is a shell built-in, and has access to the shell variables. Thus it can be used to not only access, but also alter the variables.

```
$ a=5
$ echo $a
5
$ let a++
$ echo $a
6
```

1.3 Removing Variables

To remove a variable, we use the `unset` command.

```
$ var="Hello World"
$ echo $var
Hello World
$ unset var
$ echo $var
```

This can also be used to remove multiple variables.

```
$ var1="Hello"
$ var2="World"
$ echo $var1 $var2
Hello World
$ unset var1 var2
$ echo $var1 $var2
```

Variables can also be unset by setting them to an empty string.

```
$ var="Hello World"
$ echo $var
Hello World
$ var=""
$ echo $var
```

However, this does not remove the variable, but only sets it to an empty string.

The difference between unsetting a variable and setting it to an empty string is that the unset variable is not present in the shell, while the variable set to an empty string is present in the shell. This can be observed using the `test` shell built-in.

The shell built-in version of `test` command can detect if a variable is set or not using the `-v` flag, this is not present in the executable version since an external executable can not read the shell variables.

```
$ var="Hello World"
$ echo $var
Hello World
$ test -v var ; echo $?
0
$ var=""
$ echo $var

$ test -v var ; echo $?
0
$ unset var
$ echo $var

$ test -v var ; echo $?
1
```

1.4 Listing Variables

1.4.1 set

To list all the variables in the shell, we use the `set` command. This displays all the variables and functions defined in the shell.

The output of the `set` command is very long, and we can use the `less` command to scroll through it. Here we are showing random 10 lines from the output.

```
$ set | head -n120 | shuf | head
BASH=/bin/bash
LC_NUMERIC=en_US.UTF-8
LESS_TERMCAP_so=$'\E[01;44;33m'
BASH_REMATCH=()
PATH=/opt/google-cloud-cli/bin:/sbin:/bin:/usr/local/sbin:/usr/
    local/bin:/usr/bin:/usr/sbin:/opt/android-sdk/cmdline-tools/
    latest/bin:/opt/android-sdk/platform-tools:/opt/android-sdk/
    tools:/opt/android-sdk/tools/bin:/usr/lib/jvm/default/bin:/
    usr/bin/site_perl:/usr/bin/vendor_perl:/usr/bin/core_perl:/
    usr/lib/rustup/bin:/home/sayan/scripts:/home/sayan/.android/
    sdk:/home/sayan/.android/sdk/tools:/home/sayan/.android/sdk/
    platform-tools:/home/sayan/scripts:/home/sayan/.local/bin:/
    home/sayan/.pub-cache/bin:/usr/lib/jvm/default/bin:/home/
    sayan/.fzf/bin
```

```
HOSTTYPE=x86_64
COLUMNS=187
SHELL=/bin/bash
BROWSER=thorium-browser
```

`set` also lists the user-defined variables defined in the shell.

```
$ var1=5
$ set | grep var1
var1=5
```

1.4.2 declare

Another way to list the variables is to use the `declare` command. It lists all the environment variables, shell variables, and functions defined in the shell.

```
$ declare | head -n10
ANDROID_HOME=/home/sayan/.android/sdk
ANDROID_SDK_ROOT=/home/sayan/.android/sdk
AWT_TOOLKIT=MToolkit
BASH=/bin/bash
BASHOPTS=autocd:cdspell:checkjobs:checkwinsize:cmdhist:
    complete_fullquote:execfail:expand_aliases:extglob:extquote:
    force_fignore:globasciiranges:globskipdots:histappend:
    interactive_comments:patsub_replacement:progcomp:promptvars:
    sourcepath
BASH_ALIASES=()
BASH_ARGC=([0]="0")
BASH_ARGV=()
BASH_CMDS=()
BASH_COMPLETION_VERSINFO=([0]="2" [1]="11")
```

It can also be used to list the user-defined variables.

```
$ var1=5
$ declare | grep var1
var1=5
```

1.4.3 env

Although the `env` command is used to run a command in a modified environment, it can also be used to list all the environment variables if no arguments are supplied to it.

```
$ env | head -n10
SHELL=/bin/bash
WINDOWID=100663324
COLORTERM=truecolor
LANGUAGE=
LC_ADDRESS=en_US.UTF-8
JAVA_HOME=/usr/lib/jvm/default
LC_NAME=en_US.UTF-8
SSH_AUTH_SOCK=/run/user/1000/ssh-agent.socket
SHELL_SESSION_ID=91c0e4dcd4b644e8bfa2a25613b60f60
XDG_CONFIG_HOME=/home/sayan/.config
```

This only lists the environment variables, and not the shell variables. Only if the shell variable is exported, it will be listed in the output of the `env` command.

```
$ var1=5
$ env | grep var1
$ export var1
$ env | grep var1
var1=5
```

`env` is an external command and not a shell built-in, thus it does not have the access to unexported shell variables at all.

1.4.4 printenv

The `printenv` command is used to print all the environment variables. Similar to the `env` command, it only lists the environment variables, and not the shell variables since it is an executable and not a shell built-in.

```
$ printenv | shuf | head
KONSOLE_VERSION=240202
FZF_DEFAULT_COMMAND=fd --type f -H
HOME=/home/sayan
LANGUAGE=
KONSOLE_DBUS_WINDOW=/Windows/1
USER=sayan
CLOUDSDK_PYTHON=/usr/bin/python
COLORFGBG=15;0
VISUAL=nvim
SSH_AUTH_SOCK=/run/user/1000/ssh-agent.socket
```

1.5 Special Variables

The bash shell exports some special variables whose values are set by the shell itself. These are useful to refer in scripts, and also to understand the environment in which the script is running. They let the script to be more dynamic and adapt to the environment.

- `USER` stores the currently logged in user. [2]
- `HOME` stores the home directory of the user.
- `PWD` stores the current working directory.
- `SHELL` stores the path of the shell being used.
- `PATH` stores the paths to search for commands.
- `PS1` stores the prompt string for the shell.
- `PS2` stores the secondary prompt string for the shell
- `HOSTNAME` stores the network name of the system
- `OSTYPE` stores the type of operating system.
- `TERM` stores the terminal type.

2: Originally, the **System V** distributions exported the `LOGNAME` variable, while the **BSD** distributions exported the `USER` variable. Modern distros export both, but `USER` is more commonly used. The **zsh** shell exports `USERNAME` as well.

The shell also sets some special variables that are useful in scripts. These are not exported, but set for every shell or child process accordingly.

- `$0` stores the name of the script or shell.
- `$1, $2, $3, ...` store the arguments to the script.
- `$#` stores the number of arguments to the script.

- `$*` stores all the arguments to the script as a single string.
- `$@` stores all the arguments to the script as array of strings.
- `$?` stores the exit status of the last command.
- `$$` stores the process id of the current shell.
- `$!` stores the process id of the last background command.
- `$-` stores the current options set for the shell.
- `$IFS` stores the Internal Field Separator.
- `$LINENO` stores the current line number of the script.
- `$RANDOM` stores a random number.
- `$SECONDS` stores the number of seconds the script has been running.

These variables are automatically set and updated by the shell whenever required.

The `PWD` variable is updated whenever the current working directory changes.

1.5.1 PWD

```
$ echo $PWD
/home/sayan
$ cd /tmp
$ echo $PWD
/tmp
```

The `RANDOM` variable stores a random number between 0 and 32767, and is constantly changed.

1.5.2 RANDOM

```
$ echo $RANDOM
11670
$ echo $RANDOM
29897
```

Remark 1.5.1 The `RANDOM` variable is not truly random, but is a pseudo-random number generated by the shell. It is generated using the Linear Congruential Generator algorithm. The seed for the random number is the process id of the shell. Thus, the same sequence of random numbers will be generated if the shell is restarted. To get a more cryptographically secure random number, we can use the `openssl` command or read from the `/dev/urandom` file. Read more here.

1.5.3 PATH

The `PATH` variable stores the paths to search for commands. Whenever a command is executed in the shell, the shell searches for the command in the directories listed in the `PATH` variable if it is not a shell keyword or a shell built-in. If an executable with that name with execute permissions is not found in any of the paths mentioned in `PATH` variable, then the command fails.

The `PATH` variable is colon separated. It is not set automatically, rather it has to be set by the system. It is usually set in the `/etc/profile` file and the `~/.bashrc` file.

```
$ echo "echo hello" > sayhello
$ chmod 764 sayhello
mode of 'sayhello' changed from 0644 (rw-r--r--) to 0764 (rwxrw-r
    --)
$ sayhello
bash: sayhello: command not found
$ PATH=$PATH:$PWD
$ sayhello
hello
```

Here we create an executable that prints `hello` to the terminal. It is present in our current directory, but the shell does not know where to find it, as the current directory is not present in the `PATH` variable. Once we add the current directory to the `PATH` variable, the shell is able to find the executable and execute it.

1.5.4 PS1

The `PS1` variable stores the primary prompt string for the shell. It can be used to customize the prompt of the shell.

```
[sayan@rex ~] $ PS1="[\u@\h \w] \$ "
[sayan@rex ~] $ PS1="hello "
hello PS1="enter command> "
enter command> PS1="User: \u> "
User: sayan> PS1="User: \u, Computer: \h> "
User: sayan, Computer: rex> PS1="Date: \d> "
Date: Thu Jul 25> PS1="Time: \t> "
Time: 17:35:44> PS1="Jobs: \j> "
Jobs: 0> sleep 50 &
[1] 3600280
Jobs: 1>
Jobs: 1> PS1="Shell: \s> "
Shell: bash> PS1="History Number: \!> "
[1]+  Done                    sleep 50
History Number: 563>
History Number: 563> echo hello
hello
History Number: 564> PS1="Command Number: \#> "
Command Number: 65>
Command Number: 65> echo hello
hello
Command Number: 66> PS1="Ring a bell \a> "
Ring a bell >
Ring a bell > PPS1="[\u@\h \w] \$ "
[sayan@rex ~] $
```

These changes are temporary and are only valid for the current shell session. To make the changes permanent, we need to add the `PS1` variable assignment to the `~/.bashrc` file.

The `PS1` variable gives us some customization, however it is limited. To run any arbitrary command to determine the prompt, we can use the `PROMPT_COMMAND` variable.

For example, if you want to display the exit code of the last command in the prompt, you can use the following command.

Notice how the exit code of the last command is displayed in the prompt and is updated whenever a new command is run.

```
$ prompt(){
> PS1="($?)[\u@\h \w]\$ "
> }
$ PROMPT_COMMAND=prompt
(0)[sayan@rex ~]$ ls /home
sayan
(0)[sayan@rex ~]$ ls /random
ls: cannot access '/random': No such file or directory
(2)[sayan@rex ~]$
```

We can also show the exit code of each process in the prompt if piped commands are used.

```
$ export PROMPT_COMMAND="
       _RES=\${PIPESTATUS[*]};
       _RES_STR='';
       for res in \$_RES; do
         if [[ ( \$res > 0 ) ]]; then
           _RES_STR=\" [\$_RES]\";
         fi;
       done"
$ export PS1="\u@\h \w\$_RES_STR\\$ "
sayan@rex ~$ echo hello
hello
sayan@rex ~$ exit 1 | exit 2 | exit 3
sayan@rex ~ [1 2 3]$
```

Read more here.

We can also color the prompt using ANSI escape codes. We will these in details in later chapters.

1.6 Variable Manipulation

1.6.1 Default Values

The `:-` operator is used to substitute a default value if the variable is not set or is empty. This simply returns the value, and the variable still remains unset.

```
$ unset var
$ echo ${var:-default}
default
$ echo $var

$ var=hello
$ echo ${var:-default}
hello
$ echo $var
hello
```

The `:+` operator is used to substitute a replacement value if the variable is set, but not do anything if not present.

```
$ unset var
$ echo ${var:+default}

$ var=hello
$ echo ${var:+default}
default
```

```
$ echo $var
hello
```

The `:=` operator is used to substitute a default value if the variable is not set or is empty, and also set the variable to the default value. [3] This is similar to the `:-` operator, but also sets the variable to the default value. It does nothing if the variable is already set.

3: This operator is also present in modern python and is called the walrus operator.

```
$ unset var
$ echo ${var:=default}
default
$ echo $var
default
$ var=hello
$ echo ${var:=default}
hello
$ echo $var
hello
```

These operations consider an empty variable as unset. However, sometimes we may need to consider an empty variable as set, but empty. In those cases, we can drop the colon from the operator.

Remark 1.6.1 `echo ${var:-default}` will print `default` if `var` is absent **or** empty.
`echo \${var-default}` will print `default` if `var` is absent, but not if `var` is empty.
`echo \${var:+default}` will print `default` if `var` is present **and** not empty.
`echo \${var+default}` will print `default` if `var` is present, regardless of whether `var` is empty or not.

1.6.2 Error if Unset

Sometimes we may want to throw an error if a variable is unset. This is useful in scripts to ensure that all the required variables are set when performing critical operations.

Imagine the following line of code.

```
$ rm -rf "$STEAMROOT/"
```

This looks like a simple line to remove the steam root directory. However, if the `STEAMROOT` variable is unset, this will expand to `rm -rf /`, which will delete the root directory of the system if ran with priviledge, or at the very least, the entire home directory of the user. This is a very dangerous operation, and can lead to loss of data.

The reason for the specific example is that this is a bug that was present in the steam installer script, and was fixed by Valve after it was reported.

To prevent this, we can use the `:?` operator to throw an error if the variable is unset.

```
$ unset STEAMROOT
$ rm -rf "${STEAMROOT:?Variable not set}/"
bash: STEAMROOT: Variable not set
```

The shell will not even attempt to run the command if the variable is unset, and will throw an error immediately, saving the day.

1.6.3 Length of Variable

The length of a variable can be found using the `#` operator.

```
$ var="Hello World"
$ echo ${#var}
11
```

1.6.4 Substring of Variable

The substring of a variable can be found using the `:` operator.

The syntax is `${var:start:length}`. The `start` is the index of the first character of the substring, and the `length` is the number of characters to include in the substring. The index starts from zero. If the length exceeds the end of the string, it will print till the end and not throw any error. The index can also be negative, in which case it is counted from the end of the string, similar to Python. [4]

4: In case of a negative index, the space between the colon and the negative index is important. If there is no space, it will be considered as a default substitution.

```
$ var="Hello World"
$ echo ${var:0:5}
Hello
$ echo ${var:6:5}
World
$ echo ${var:6:50}
World
$ echo ${var: -5:5}
World
$ echo ${var: -11:5}
Hello
```

1.6.5 Prefix and Suffix Removal

The prefix and suffix of a variable can be removed using the `#` and `\%` operators respectively. Any glob like pattern can be matched, not just fixed strings.

The match can be made greedy (longest match) by using `##` and `\%\%` respectively. This applies for wildcard matching.

- `${var%pattern}` will delete the shortest match of `pattern` from the end of `var`.
- `${var%%pattern}` will delete the longest match of `pattern` from the end of `var`.
- `${var#pattern}` will delete the shortest match of `pattern` from the start of `var`.
- `${var##pattern}` will delete the longest match of `pattern` from the start of `var`.

Here the dot is used as a separator, and we want to extract the first and last part of the string. The dot does not signify a wildcard, but a literal dot, since this is not regex.

If we have a variable with value "abc.def.ghi.xyz".

- `echo ${var%.*}` will print "abc.def.ghi".

- `echo ${var%%.*}` will print "abc".
- `echo ${var#*.}` will print "def.ghi.xyz".
- `echo ${var##*.}` will print "xy

1.6.6 Replace Substring

The substring of a variable can be replaced using the `/` operator. The syntax is `${var/pattern/string}`. This will replace the first occurence of `pattern` with `string`. The search pattern can be a glob pattern, not just a fixed string. The replacement has to be a fixed string, and not a glob pattern.

```
$ var="Hello World"
$ echo ${var/ */ Universe}
Hello Universe
```

We can also replace all occurences of the pattern using the `//` operator.

```
$ var="Hello World"
$ echo ${var//o/O}
HellO WOrld
```

1.6.7 Anchoring Matches

We can also anchor the match to the start or end of the string using the `#` and `\%` operators respectively.

Observe how the `#` operator anchors the match to the start of the string. Even though the second variable also had an `s` in the middle, it was not replaced.

```
$ var="system commands"
$ echo ${var/#s/S}
System commands
$ var="linux commands"
$ echo ${var/#s/S}
linux commands
```

Similarly, we can anchor the match to the end of the string using the `\%` operator.

```
$ var="Hello World"
$ echo ${var/%?/&!}
Hello World!
```

Here we are using the `?` wildcard to match any character, and replace it with `&!`. The `&` is used to refer to the matched string and is not interpreted as a literal ampersand.

1.6.8 Deleting the match

We can also delete the match by keeping the replacement string empty.

```
$ var="Hello World"
$ echo ${var/% */}
```

This matches all the words after the first word, and deletes them. Here the match is always greedy, and will match the longest possible string.

1.6.9 Lowercase and Uppercase

The case of a variable can be changed using the , and ^ operators. This changes only the first character of the variable. To change the entire variable, we can use the ,, and ^^ operators.

```
$ var="sayan"
$ echo ${var^}
Sayan
```

Similarly, we can change the entire variable.

```
$ var="SAYAN"
$ echo ${var,,}
sayan
```

This is useful if you want to approximate the user's name from the username.

```
$ echo "Hello ${USER^}!"
Hello Sayan!
```

1.6.10 Sentence Case

To convert the first letter of a variable to uppercase, and the rest to lowercase, we can use the following command.

```
var="hELLO wORLD"
lower=${var,,}
echo ${lower^}
Hello world
```

Here we are simply using the two operators in sequence to achieve the desired result.

1.7 Restrictions on Variables

Since bash variables are untyped, they can be set to any value. However, sometimes we may want to restrict the type of value that can be stored in a variable.

This can be done using the `declare` command.

1.7.1 Integer Only

To restrict a variable to only store integers, we can use the `-i` flag.

```
$ declare -i ‘var‘
$ var=5
$ echo "$var * $var = $((var**2))"
5 * 5 = 25
$ var=hello
$ echo "$var * $var = $((var**2))"
0 * 0 = 0
```

If we assign any non-integer value to the variable, it will be treated as zero. This will not throw an error, but will silently set the variable to zero.

1.7.2 No Upper Case

To automatically convert all the characters of a variable to lowercase, we can use the `-l` flag. This does not change non-alphabetic characters.

```
$ declare -l var
$ var="HELLO WORLD!"
$ echo $var
hello world!
```

1.7.3 No Lower Case

Similarly, we can use the `-u` flag to convert all the characters of a variable to uppercase, while retaining non-alphabetic characters as-is.

```
$ declare -u var
$ var="hello world!"
$ echo $var
HELLO WORLD!
```

1.7.4 Read Only

To make a variable read only, we can use the `-r` flag. This means we cannot change the value of the variable once it is set. Thus the value also has to be set at the time of declaration. [5]

5: This way of assigning the value is also possible for the other flags, but is not necessary.

```
$ declare -r PI=3.14159
$ echo "PI = $PI"
PI = 3.14159
$ PI=3.1416
-bash: PI: readonly variable
```

1.7.5 Removing Restrictions

We can remove the restrictions from a variable using the + flag. This cannot be done for the read only flag.

```
$ declare -i var
$ var=hello
$ echo $var
0
$ declare +i var
$ var=hello
$ echo $var
hello
```

1.8 Bash Flags

There are some flags that can be set in the bash shell to change the behaviour of the shell. The currently set flags can be viewed using the `echo \$-` command.

```
$ echo $-
himBHs
```

These can be set or unset using the `set` command.

```
$ echo $-
himBHs
$ set +m
$ echo $-
hiBHs
```

The same convention as the `declare` command is used, with `+` to unset the flag, and `-` to set the flag.

The default flags are:

- **h**: locate and remember (hash) commands as they are looked up.
- **i**: **interactive** shell
- **m**: monitor the jobs and report changes
- **B**: **braceexpand** - expand the expression in braces
- **H**: **histexpand** - expand the history command
- **s**: Read commands from the standard input.

We can see the default flags change if we start a non-interactive shell.

```
$ bash -c 'echo $-'
hBc
```

The **c** flag means that `bash` is reading the command from the argument and it assigns `$0` to the first non-option argument.

Some other important flags are:

- **e**: Exit immediately if a command exits with a non-zero status.
- **u**: Treat unset variables as an error when substituting.
- **x**: Print commands and their arguments as they are executed.
- **v**: Print shell input lines as they are read.
- **n**: Read commands but do not execute them. This is useful for testing syntax of a command.
- **f**: Disable file name generation (globbing).
- **C**: Also called **noclobber**. Prevent overwriting of files using redirection.

If we set the **-f** flag, the shell will not expand the glob patterns which we discussed in Chapter **??**.

1.9 Signals

Remark 1.9.1 If you press `Ctrl+S` in the terminal, some terminals might stop responding. You can resume it by pressing `Ctrl+Q`. This is because `ixoff` is set. You can unset it using `stty -ixon`. This is a

common problem with some terminals, thus it can placed inside the `~/.bashrc` file.

Other Ctrl+Key combinations:

- **Ctrl+C**: Interrupt the current process, this sends the SIGINT signal.
- **Ctrl+D**: End of input, this sends the EOF signal.
- **Ctrl+L**: Clear the terminal screen.
- **Ctrl+Z**: Suspend the current process, this sends the SIGTSTP signal.
- **Ctrl+R**: Search the history of commands using reverse-i-search.
- **Ctrl+T**: Swap the last two characters
- **Ctrl+U**: Cut the line before the cursor
- **Ctrl+V**: Insert the next character literally
- **Ctrl+W**: Cut the word before the cursor
- **Ctrl+Y**: Paste the last cut text

1.10 Brace Expansion

As the `B` flag is set by default, brace expansion is enabled in the shell.

Definition 1.10.1 (Brace Expansion) Brace expansion is a mechanism by which arbitrary strings can be generated using a concise syntax. It is similar to pathname expansion, but can be used to expand to non-existing patterns too.

Brace expansion is used to generate list of strings, and is useful in generating sequences of strings.

1.10.1 Range Expansion

For generating a sequence of numbers, we can use the range expansion.

Syntax:

```
{start...end}
```

```
$ echo {1..5}
1 2 3 4 5
```

We can also specify the increment value.

```
$ echo {1..11..2}
1 3 5 7 9 11
```

The start and end values are both inclusive.

This can also be used for alphabets.

```
$ echo {a..f}
a b c d e f
```

1.10.2 List Expansion

For generating a list of strings, we can use the list expansion.

Syntax:

```
{string1,string2,string3}
```

This can be used to generate a list of strings.

```
$ echo {apple,banana,cherry}
apple banana cherry
```

1.10.3 Combining Expansions

The real power of brace expansion comes when we combine the expansion with a static part.

```
$ echo file{1..5}.txt
file1.txt file2.txt file3.txt file4.txt file5.txt
```

Brace expansion automatically expands to the cartesian product of the strings if multiple expansions are present in a single token.

```
$ echo {a,b}{1,2}
a1 a2 b1 b2
```

We can also combine multiple tokens with space in between by escaping the space.

```
$ echo {a,b}\ {1,2}
a 1 a 2 b 1 b 2
```

The expansion is done from left to right, and the order of the tokens is preserved.

This can be used to create a list of files following some pattern.

Here we are using ascii encoded output of the `tree` command for compatibility with the book. Feel free to drop the `--charset ascii` when trying this out in your terminal.

```
$ mkdir -p test/{a,b,c}/{1,2,3}
$ touch test/{a,b,c}/{1,2,3}/file{1..5}.txt
$ tree --charset ascii test
test
|-- a
|   |-- 1
|   |   |-- file1.txt
|   |   |-- file2.txt
|   |   |-- file3.txt
|   |   |-- file4.txt
|   |   `-- file5.txt
|   |-- 2
|   |   |-- file1.txt
|   |   |-- file2.txt
|   |   |-- file3.txt
|   |   |-- file4.txt
|   |   `-- file5.txt
|   `-- 3
|       |-- file1.txt
|       |-- file2.txt
|       |-- file3.txt
|       |-- file4.txt
|       `-- file5.txt
```

```
|-- b
|   |-- 1
|   |   |-- file1.txt
|   |   |-- file2.txt
|   |   |-- file3.txt
|   |   |-- file4.txt
|   |   `-- file5.txt
|   |-- 2
|   |   |-- file1.txt
|   |   |-- file2.txt
|   |   |-- file3.txt
|   |   |-- file4.txt
|   |   `-- file5.txt
|   `-- 3
|       |-- file1.txt
|       |-- file2.txt
|       |-- file3.txt
|       |-- file4.txt
|       `-- file5.txt
`-- c
    |-- 1
    |   |-- file1.txt
    |   |-- file2.txt
    |   |-- file3.txt
    |   |-- file4.txt
    |   `-- file5.txt
    |-- 2
    |   |-- file1.txt
    |   |-- file2.txt
    |   |-- file3.txt
    |   |-- file4.txt
    |   `-- file5.txt
    `-- 3
        |-- file1.txt
        |-- file2.txt
        |-- file3.txt
        |-- file4.txt
        `-- file5.txt

13 directories, 45 files
```

1.11 History Expansion

Definition 1.11.1 (History Expansion) History expansion is a mechanism by which we can refer to previous commands in the history list.

It is enabled using the `H` flag.

We can run `history` to see the list of commands in the history. To re-run a command, we can use the `!` operator along with the history number.

```
$ echo hello
hello
$ history | tail
  499  man set
```

```
  500  man set
  501  man tree
  502  tree --charset unicode
  503  tree --charset ascii
  504  history
  505  tree --charset unicode
  506  clear
  507  echo hello
  508  history | tail
$ !507
echo hello
hello
```

The command itself is output to the terminal before it is executed.

We can also refer to the last command using the `!!` operator.

```
$ touch file1 file2 .hidden
$ ls
file1  file2
$ !! -a
ls -a
.  ..  .hidden  file1  file2
```

One frequent use of history expansion is to run a command as root. If we forget to run a command as root, we can use the `sudo` command to run the last command as root.

```
$ touch /etc/test
touch: cannot touch '/etc/test': Permission denied
$ sudo !!
sudo touch /etc/test
```

1.12 Arrays

Definition 1.12.1 (Array) An array is a collection of elements, each identified by an index.

We can declare an array using the `declare` command.

```
$ declare -a arr
```

However, this is not necessary, as bash automatically creates an array when we assign multiple values to a variable.

```
$ arr=(1 2 3 4 5)
$ echo ${arr[2]}
3
```

We can set the value of each element of the array using the index.

```
$ arr[2]=6
$ echo ${arr[2]}
6
```

If we only access the variable without the index, it will return the first element of the array.

```
$ arr=(1 2 3 4 5)
$ echo $arr
1
```

1.12.1 Length of Array

The length of the array can be found using the `\#` operator.

```
$ arr=(1 2 3 4 5)
$ echo ${#arr[@]}
```

1.12.2 Indices of Array

Although it looks like a continuous sequence of numbers, the indices of the array are not necessarily continuous. Bash arrays are actually dictionaries or hash-maps and the index is the key. Thus we may also need to get the indices of the array.

```
$ arr=(1 2 3 4 5)
$ arr[10]=6
$ echo ${!arr[@]}
0 1 2 3 4 10
```

1.12.3 Printing all elements of Array

To print all the elements of the array, we can use the `@` operator.

By default, the indices start from zero and are incremented by one.

```
$ arr=(1 2 3 4 5)
$ echo ${arr[@]}
1 2 3 4 5
```

In indexed arrays, the indices are always integers, however, if we try to use a non-integer index, it will be treated as zero.

```
$ arr=(1 2 3 4 5)
$ arr["hello"]=6
$ echo ${arr[@]}
6 2 3 4 5
```

1.12.4 Deleting an Element

To delete an element of an array, we can use the `unset` command.

```
$ arr=(1 2 3 4 5)
$ arr[10]=6
$ echo ${arr[@]}
1 2 3 4 5 6
$ echo ${!arr[@]}
0 1 2 3 4 10
$ unset arr[10]
$ echo ${arr[@]}
1 2 3 4 5
$ echo ${!arr[@]}
0 1 2 3 4
```

1.12.5 Appending an Element

If we want to simply append an element to the array without specifying the index, we can use the += operator. The index of the new element will be the next integer after the last element.

```
$ arr=(1 2 3 4 5)
$ arr[10]=6
$ arr+=(7)
$ echo ${arr[@]}
1 2 3 4 5 6 7
$ echo ${!arr[@]}
0 1 2 3 4 10 11
```

We can also append multiple elements at once by separating them with spaces.

1.12.6 Storing output of a command in an Array

We can store the output of a command in a normal variable using the = operator.

```
$ var=$(ls)
$ echo $var
file1 file2 file3
```

But if we want to iterate over the output of a command, we can store it in an array by surrounding the command evaluation with parenthesis.

```
$ arr=($(ls))
$ echo ${arr[@]}
file1 file2 file3
$ echo ${!arr[@]}
0 1 2
$ echo ${arr[1]}
file2
```

1.12.7 Iterating over an Array

We can iterate over an array using a `for` loop.

```
$ arr=(1 2 3 4 5)
$ for i in ${arr[@]}; do
>   echo $i
> done
1
2
3
4
5
```

We will cover loops in more detail in Chapter **??**.

Different Ways of Iterating

There are three ways to iterate over an array depending on how we break the array.

Treat entire array as a single element

```
$ arr=("Some" "elements" "are" "multi word")
$ for i in "${arr[*]}"; do
>   echo $i
> done
Some elements are multi word
```

Here, as we are using the * operator, the array is expandded as string, and not array elements. Further, as we have then quoted the variable, the for loop does not break the string by the spaces.

Break on each word

This can be done in two ways, either by using the `@` operator, or by using the * operator. In either case we do not quote the variable.

```
$ arr=("Some" "elements" "are" "multi word")
$ for i in ${arr[*]}; do    echo $i;  done
Some
elements
are
multi
word
```

```
$ arr=("Some" "elements" "are" "multi word")
$  for i in ${arr[@]}; do    echo $i;  done
Some
elements
are
multi
word
```

Break on each element

Finally, the last way is to break on each element of the array. This is often the desired way to iterate over an array. We use the `@` operator, and quote the variable to prevent word splitting.

```
$ arr=("Some" "elements" "are" "multi word")
$  for i in "${arr[@]}"; do    echo $i;  done
Some
elements
are
multi word
```

1.13 Associative Arrays

If we want to store key-value pairs, we can use associative arrays. In this, the index is not an integer, but a string. It also does not automatically assign the next index, but we have to specify the index.

We use the `declare -A` command to declare an associative array, although this is not necessary.

```
$ declare -A arr
$ arr=(["name"]="Sayan" ["age"]=22)
$ echo ${arr[name]}
Sayan
$ arr["age"]=23
$ echo ${#arr[@]}
2
$ echo ${!arr[@]}
age name
$ echo ${arr[@]}
23 Sayan
$ unset arr[name]
```

In bash, the order of the elements is not preserved, and the elements are not sorted. This was how Python dictionaries worked before Python 3.7.