

# **Navigating Linux**

**A guide for beginners to the Shell and GNU coreutils**

Sayan Ghosh

January 12, 2025

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 , Sushil Pachpinde , and Subendu Gorai**

## **Copyright**

© This book is released under the public domain, meaning it is freely available for use and distribution without restriction. However, while the content itself is not subject to copyright, it is requested that proper attribution be given if any part of this book is quoted or referenced. This ensures recognition of the original authorship and helps maintain transparency in the dissemination of information.

## **Colophon**

This document was typeset with the help of **KOMA-Script** and **LAT<sub>E</sub>X** 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 January 12, 2025

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

Preface	v
Contents	vii
<b>1 Essentials of Linux</b>	<b>1</b>
1.1 Introduction . . . . .	1
1.1.1 What is Linux? . . . . .	1
1.1.2 Desktop Environments . . . . .	4
1.1.3 Window Managers . . . . .	5
1.1.4 Why Linux? . . . . .	6
1.1.5 What is Shell? . . . . .	7
1.1.6 Shell vs Terminal . . . . .	8
1.1.7 Why the Command Line? . . . . .	10
1.1.8 Command Prompt . . . . .	11
1.2 Simple Commands in GNU Core Utils . . . . .	12
1.2.1 File System Navigation . . . . .	12
1.2.2 Manuals . . . . .	16
1.2.3 System Information . . . . .	20
1.2.4 File Management . . . . .	24
1.2.5 Text Processing and Pagers . . . . .	29
1.2.6 Aliases and Types of Commands . . . . .	35
1.2.7 User Management . . . . .	41
1.2.8 Date and Time . . . . .	43
1.3 Navigating the File System . . . . .	47
1.3.1 What is a File System? . . . . .	47
1.3.2 In Memory File System . . . . .	51
1.3.3 Paths . . . . .	56
1.3.4 Basic Commands for Navigation . . . . .	57
1.4 File Permissions . . . . .	60
1.4.1 Read . . . . .	62
1.4.2 Write . . . . .	62
1.4.3 Execute . . . . .	62

1.4.4	Interesting Caveats . . . . .	63
1.4.5	Changing Permissions . . . . .	65
1.4.6	Special Permissions . . . . .	66
1.4.7	Octal Representation of Permissions . . . . .	70
1.5	Types of Files . . . . .	73
1.5.1	Regular Files . . . . .	73
1.5.2	Directories . . . . .	73
1.5.3	Symbolic Links . . . . .	73
1.5.4	Character Devices . . . . .	74
1.5.5	Block Devices . . . . .	75
1.5.6	Named Pipes . . . . .	76
1.5.7	Sockets . . . . .	77
1.5.8	Types of Regular Files . . . . .	78
1.6	Inodes and Links . . . . .	80
1.6.1	Inodes . . . . .	80
1.6.2	Separation of Data, Metadata, and Filename . . . . .	82
1.6.3	Directory Entries . . . . .	83
1.6.4	Hard Links . . . . .	84
1.6.5	Symbolic Links . . . . .	86
1.6.6	Symlink vs Hard Links . . . . .	89
1.6.7	Identifying Links . . . . .	89
1.6.8	What are . and ..? . . . . .	90
<b>2</b>	<b>Command Line Editors</b>	<b>93</b>
2.1	Introduction . . . . .	93
2.1.1	Types of Editors . . . . .	93
2.1.2	Why Command Line Editors? . . . . .	94
2.1.3	Mouse Support . . . . .	94
2.1.4	Editor war . . . . .	95
2.1.5	Differences between Vim and Emacs . . . . .	95
2.1.6	Nano: The peacemaker amidst the editor war . . . . .	98
2.2	Vim . . . . .	98
2.2.1	History . . . . .	98
2.2.2	Ed Commands . . . . .	110
2.2.3	Exploring Vim . . . . .	121

2.3	Emacs . . . . .	133
2.3.1	History . . . . .	133
2.3.2	Exploring Emacs . . . . .	137
2.4	Nano . . . . .	139
2.4.1	History . . . . .	139
2.4.2	Exploring Nano . . . . .	141
2.4.3	Editing A Script in Nano . . . . .	142
<b>3</b>	<b>Networking and SSH</b>	<b>145</b>
3.1	Networking . . . . .	145
3.1.1	What is networking? . . . . .	145
3.1.2	Types of Networks . . . . .	146
3.1.3	Devices in a Network . . . . .	147
3.1.4	IP Addresses . . . . .	149
3.1.5	Subnetting . . . . .	150
3.1.6	Private and Public IP Addresses . . . . .	153
3.1.7	CIDR . . . . .	154
3.1.8	Ports . . . . .	154
3.1.9	Protocols . . . . .	156
3.1.10	Firewalls . . . . .	157
3.1.11	SELinux . . . . .	158
3.1.12	Network Tools . . . . .	160
3.2	SSH . . . . .	169
3.2.1	What is SSH? . . . . .	169
3.2.2	History . . . . .	169
3.2.3	How does SSH work? . . . . .	170
3.2.4	Key-based Authentication . . . . .	170
3.2.5	Configuring your SSH keys . . . . .	171
3.2.6	Sharing your public key . . . . .	173
3.2.7	How to login to a remote server . . . . .	174
3.2.8	Call an exorcist, there's a daemon in my computer . . . . .	175
3.2.9	SCP . . . . .	179
<b>4</b>	<b>Process Management</b>	<b>181</b>
4.1	What is sleep? . . . . .	181
4.1.1	Example . . . . .	181

4.1.2	Scripting with sleep . . . . .	181
4.1.3	Syntax and Synopsis . . . . .	182
4.2	Different ways of running a process . . . . .	183
4.2.1	What are processes? . . . . .	183
4.2.2	Process Creation . . . . .	184
4.2.3	Process Ownership . . . . .	184
4.2.4	Don't kill my children . . . . .	186
4.2.5	Setsid . . . . .	188
4.2.6	Nohup . . . . .	190
4.2.7	coproc . . . . .	190
4.2.8	at and cron . . . . .	192
4.2.9	GNU parallel . . . . .	192
4.2.10	systemd services . . . . .	192
4.3	Process Management . . . . .	193
4.3.1	Disown . . . . .	193
4.3.2	Jobs . . . . .	194
4.3.3	Suspending and Resuming Jobs . . . . .	195
4.3.4	Killing Processes . . . . .	197
4.4	Finding Processes . . . . .	202
4.4.1	pgrep . . . . .	202
4.4.2	pkill . . . . .	202
4.4.3	pidwait . . . . .	203
4.5	Listing Processes . . . . .	204
4.5.1	ps . . . . .	204
4.5.2	pstree . . . . .	205
4.5.3	top . . . . .	206
4.5.4	htop . . . . .	207
4.5.5	btop . . . . .	208
4.5.6	glances . . . . .	209
4.6	Exit Codes . . . . .	210
<b>5</b>	<b>Streams, Redirections, Piping</b>	<b>213</b>
5.1	Multiple Commands in a Single Line . . . . .	213
5.1.1	Conjunction and Disjunction . . . . .	213
5.2	Streams . . . . .	218

5.3	Redirection . . . . .	220
5.3.1	Standard Output Redirection . . . . .	221
5.3.2	Standard Error Redirection . . . . .	224
5.3.3	Appending to a File . . . . .	227
5.3.4	Standard Input Redirection . . . . .	228
5.3.5	Here Documents . . . . .	230
5.3.6	Here Strings . . . . .	231
5.4	Pipes . . . . .	232
5.4.1	UNIX Philosophy . . . . .	233
5.4.2	Multiple Pipes . . . . .	233
5.4.3	Piping Standard Error . . . . .	241
5.4.4	Piping to and From Special Files . . . . .	242
5.4.5	Named Pipes . . . . .	244
5.4.6	Tee Command . . . . .	247
5.5	Command Substitution . . . . .	248
5.6	Arithmetic Expansion . . . . .	250
5.6.1	Using variables in arithmetic expansion . . . . .	250
5.7	Process Substitution . . . . .	251
5.8	Summary . . . . .	256
<b>6</b>	<b>Pattern Matching</b>	<b>259</b>
6.1	Introduction . . . . .	259
6.2	Globs and Wildcards . . . . .	259
6.3	Regular Expressions . . . . .	263
6.3.1	Basic Regular Expressions . . . . .	264
6.3.2	Character Classes . . . . .	266
6.3.3	Anchors . . . . .	274
6.3.4	Quantifiers . . . . .	275
6.3.5	Alternation . . . . .	281
6.3.6	Grouping . . . . .	283
6.3.7	Backreferences . . . . .	284
6.4	Extended Regular Expressions . . . . .	287
6.5	Perl-Compatible Regular Expressions . . . . .	290
6.5.1	Minimal Matching (a.k.a. "ungreedy") . . . . .	291
6.5.2	Multiline matching . . . . .	291
6.5.3	Named subpatterns . . . . .	291

6.5.4	Look-ahead and look-behind assertions . . . . .	292
6.5.5	Comments . . . . .	292
6.5.6	Recursive patterns . . . . .	293
6.6	Other Text Processing Tools . . . . .	294
6.6.1	tr . . . . .	295
6.6.2	cut . . . . .	300
6.6.3	paste . . . . .	305
6.6.4	fold . . . . .	306
6.6.5	grep . . . . .	308
6.6.6	sed . . . . .	308
6.6.7	awk . . . . .	308
<b>7</b>	<b>Grep</b>	<b>309</b>
7.1	Regex Engine . . . . .	309
7.2	PCRE . . . . .	311
7.3	Print Only Matching Part . . . . .	312
7.4	Matching Multiple Patterns . . . . .	313
7.4.1	Disjunction . . . . .	313
7.4.2	Conjunction . . . . .	314
7.5	Read Patterns from File . . . . .	315
7.6	Ignore Case . . . . .	315
7.7	Invert Match . . . . .	316
7.8	Anchoring . . . . .	317
7.9	Counting Matches . . . . .	318
7.10	Print Filename . . . . .	320
7.11	Limiting Output . . . . .	321
7.12	Quiet Quitting . . . . .	324
7.13	Numbering Lines . . . . .	324
7.14	Recursive Search . . . . .	325
7.15	Context Line Control . . . . .	326
7.16	Finding Lines Common in Two Files . . . . .	327
<b>8</b>	<b>Shell Variables</b>	<b>331</b>
8.1	Creating Variables . . . . .	331
8.2	Printing Variables to the Terminal . . . . .	333
8.2.1	Echo Command . . . . .	334

8.2.2	Accessing and Updating Numeric Variables . . . . .	338
8.3	Removing Variables . . . . .	343
8.4	Listing Variables . . . . .	345
8.4.1	set . . . . .	345
8.4.2	declare . . . . .	346
8.4.3	env . . . . .	346
8.4.4	printenv . . . . .	347
8.5	Special Variables . . . . .	348
8.5.1	PWD . . . . .	349
8.5.2	RANDOM . . . . .	349
8.5.3	PATH . . . . .	350
8.5.4	PS1 . . . . .	351
8.6	Variable Manipulation . . . . .	353
8.6.1	Default Values . . . . .	353
8.6.2	Error if Unset . . . . .	354
8.6.3	Length of Variable . . . . .	355
8.6.4	Substring of Variable . . . . .	355
8.6.5	Prefix and Suffix Removal . . . . .	356
8.6.6	Replace Substring . . . . .	357
8.6.7	Anchoring Matches . . . . .	357
8.6.8	Deleting the match . . . . .	358
8.6.9	Lowercase and Uppercase . . . . .	358
8.6.10	Sentence Case . . . . .	359
8.7	Restrictions on Variables . . . . .	359
8.7.1	Integer Only . . . . .	359
8.7.2	No Upper Case . . . . .	360
8.7.3	No Lower Case . . . . .	360
8.7.4	Read Only . . . . .	360
8.7.5	Removing Restrictions . . . . .	361
8.8	Bash Flags . . . . .	361
8.9	Signals . . . . .	362
8.10	Brace Expansion . . . . .	363
8.10.1	Range Expansion . . . . .	364
8.10.2	List Expansion . . . . .	364
8.10.3	Combining Expansions . . . . .	365
8.11	History Expansion . . . . .	367

8.12	Arrays . . . . .	368
8.12.1	Length of Array . . . . .	369
8.12.2	Indices of Array . . . . .	369
8.12.3	Printing all elements of Array . . . . .	370
8.12.4	Deleting an Element . . . . .	370
8.12.5	Appending an Element . . . . .	371
8.12.6	Storing output of a command in an Array . . . . .	371
8.12.7	Iterating over an Array . . . . .	372
8.13	Associative Arrays . . . . .	373
<b>9</b>	<b>Shell Scripting</b>	<b>375</b>
9.1	What is a shell script? . . . . .	375
9.2	Shebang . . . . .	375
9.3	Comments . . . . .	377
9.3.1	Multiline Comments . . . . .	377
9.4	Variables . . . . .	378
9.5	Arguments . . . . .	380
9.5.1	Shifting Arguments . . . . .	384
9.6	Input and Output . . . . .	385
9.6.1	Reading Input . . . . .	385
9.7	Conditionals . . . . .	387
9.7.1	Test command . . . . .	387
9.7.2	Test Keyword . . . . .	391
9.8	If-elif-else . . . . .	394
9.8.1	If . . . . .	394
9.8.2	Else . . . . .	396
9.8.3	Elif . . . . .	397
9.9	Exit code inversion . . . . .	397
9.10	Mathematical Expressions as if command . . . . .	398
9.11	Command Substitution in if . . . . .	398
9.12	Switch . . . . .	398
9.12.1	Fall Through . . . . .	400
9.12.2	Multiple Patterns . . . . .	400
9.13	Select Loop . . . . .	401
9.14	Loops . . . . .	402
9.14.1	For loop . . . . .	403

9.14.2	C style for loop . . . . .	406
9.14.3	IFS . . . . .	407
9.14.4	While loop . . . . .	409
9.14.5	Until loop . . . . .	411
9.14.6	Read in while . . . . .	412
9.14.7	Break and Continue . . . . .	414
9.15	Functions . . . . .	416
9.16	Debugging . . . . .	420
9.17	Recursion . . . . .	421
9.18	Shell Arithmetic . . . . .	423
9.18.1	bc . . . . .	423
9.18.2	expr . . . . .	425
9.19	Running arbitrary commands using source, eval and exec . . . . .	427
9.19.1	exec . . . . .	428
9.20	Getopts . . . . .	428
9.20.1	With case statement . . . . .	430
9.21	Profile and RC files . . . . .	431
9.22	Summary . . . . .	432
<b>10</b>	<b>Stream Editor</b> . . . . .	<b>433</b>
10.1	Basic Usage . . . . .	433
10.2	Addressing . . . . .	434
10.2.1	Negation . . . . .	436
10.3	Commands . . . . .	436
10.3.1	Command Syntax . . . . .	436
10.3.2	Available Commands . . . . .	437
10.3.3	Branching and Flow Control . . . . .	437
10.3.4	Printing . . . . .	438
10.3.5	Deleting . . . . .	438
10.3.6	Substitution . . . . .	439
10.3.7	Print Line Numbers . . . . .	443
10.3.8	Inserting and Appending Text . . . . .	444
10.3.9	Changing Lines . . . . .	445
10.3.10	Transliteration . . . . .	446
10.4	Combining Commands . . . . .	446
10.5	In-place Editing . . . . .	448

10.6	Sed Scripts . . . . .	449
10.6.1	Order of Execution . . . . .	450
10.6.2	Shebang . . . . .	451
10.7	Branching and Flow Control . . . . .	452
10.7.1	Labels . . . . .	452
10.7.2	Branching . . . . .	453
10.7.3	Appending Lines . . . . .	456
10.7.4	Join Multiline Strings . . . . .	458
10.7.5	If-Else . . . . .	459
10.7.6	If-elif-else . . . . .	460
<b>11</b>	<b>AWK</b>	<b>461</b>
11.1	What is AWK? . . . . .	461
11.2	Basic Syntax . . . . .	462
11.2.1	Special Conditions . . . . .	463
11.2.2	Ranges . . . . .	464
11.2.3	Fields . . . . .	465
11.2.4	Number of Fields and Number of Records . . . . .	465
11.2.5	Splitting Records by custom delimiter . . . . .	466
11.3	Awk Scripts . . . . .	468
11.3.1	Command Line Arguments . . . . .	468
11.3.2	Shebang . . . . .	472
11.4	Variables . . . . .	472
11.4.1	Loose Typing and Numeric Strings . . . . .	473
11.4.2	User-defined Variables . . . . .	475
11.4.3	Built-in Variables . . . . .	479
11.5	Functions . . . . .	493
11.5.1	User-defined Functions . . . . .	494
11.5.2	Pass by Value and Pass by Reference . . . . .	497
11.5.3	Built-in Functions . . . . .	498
11.6	Arrays . . . . .	507
11.6.1	Associative Arrays . . . . .	508
11.6.2	Multidimensional Arrays . . . . .	509
11.7	Regular Expressions . . . . .	511
11.7.1	Matching Empty Regex . . . . .	513
11.7.2	Character Classes . . . . .	513

11.7.3	Regex Constant vs String Constant . . . . .	514
11.7.4	Ignoring Case . . . . .	515
11.8	Control Structures . . . . .	516
11.8.1	if-else . . . . .	517
11.8.2	Ternary Operator . . . . .	520
11.8.3	for loop . . . . .	520
11.8.4	Iterating over fields . . . . .	523
11.8.5	while loop . . . . .	524
11.8.6	do-while loop . . . . .	525
11.8.7	switch-case . . . . .	526
11.8.8	break . . . . .	531
11.8.9	continue . . . . .	531
11.8.10	next and nextfile . . . . .	532
11.8.11	exit . . . . .	535
11.8.12	BEGINFILE and ENDFILE . . . . .	536
11.9	Printing . . . . .	537
11.9.1	print . . . . .	537
11.9.2	printf . . . . .	538
11.9.3	OFS . . . . .	541
11.9.4	ORS . . . . .	542
11.10	Files and Command Line Arguments . . . . .	543
11.10.1	FILENAME . . . . .	544
11.10.2	ARGV . . . . .	545
11.11	Input using getline . . . . .	545
11.11.1	getline without arguments . . . . .	546
11.11.2	getline into variable . . . . .	547
11.11.3	getline with input file . . . . .	548
11.11.4	getline into variable from file . . . . .	549
11.11.5	getline with pipe . . . . .	551
11.11.6	getline into a variable from a pipe . . . . .	554
11.11.7	getline with coprocess . . . . .	555
11.12	Redirection and Piping . . . . .	556
11.12.1	Output Redirection . . . . .	556
11.12.2	Appending . . . . .	557
11.12.3	Piping to other commands . . . . .	557
11.12.4	Piping to a coprocess . . . . .	558

11.12.5	system	559
11.13	Examples	559
11.13.1	Finding the Frequency	559
11.13.2	Descriptive Statistics	561
11.13.3	Semi-Quoted CSV Handing	563

# List of Figures

1.1	Linux Distributions Usage . . . . .	3
1.2	Desktop Environment Usage . . . . .	5
1.3	Operating System Onion Rings . . . . .	7
1.4	GNU Core Utils Logo . . . . .	12
1.5	<code>ls -l</code> Output . . . . .	15
1.6	Linux Filesystem Hierarchy . . . . .	48
1.7	Relative Path . . . . .	57
1.8	File Permissions . . . . .	60
1.9	Octal Permissions . . . . .	70
1.10	System Calls . . . . .	82
1.11	Inode and Directory Entry . . . . .	84
1.12	Directed Acyclic Graph . . . . .	85
1.13	Abstract Representation of Symbolic Links and Hard Links . . . . .	89
1.14	Symbolic Links and Hard Links . . . . .	89
2.1	A Teletype . . . . .	100
2.2	Ken Thompson . . . . .	100
2.3	Dennis Ritchie . . . . .	101
2.4	Xerox Alto, one of the first VDU terminals with a GUI, released in 1973 . . . . .	102
2.5	A first generation Dectape (bottom right corner, white round tape) being used with a PDP-11 computer . . . . .	103
2.6	George Coulouris . . . . .	103
2.7	Bill Joy . . . . .	104
2.9	Stevie Editor . . . . .	106
2.8	The Keyboard layout of the ADM-3A terminal . . . . .	106
2.10	Bram Moolenaar . . . . .	108
2.11	The initial version of Vim, when it was called Vi IMitation . . . . .	108
2.12	Vim 9.0 Start screen . . . . .	109
2.13	Neo Vim Window editing this book . . . . .	110
2.14	Simplified Modes in Vim . . . . .	122
2.15	Detailed Modes in Vim . . . . .	123

2.16	Vim Cheat Sheet . . . . .	132
2.17	Richard Stallman - founder of GNU and FSF projects . . . . .	133
2.18	Guy L. Steele Jr. combined many divergent TECO with macros to create EMACS . . . . .	135
2.19	James Gosling - creator of Gosling Emacs and later Java . . . . .	136
2.20	ADM-3A terminal . . . . .	136
2.21	Space Cadet Keyboard . . . . .	136
2.22	Nano Text Editor . . . . .	139
3.1	Types of Networks . . . . .	146
3.2	Growth Rate of Different Functions - Note how quickly $n^2$ grows . . . . .	146
3.3	Hub and Spoke Model Employed by Airlines . . . . .	146
3.4	LAN and WAN connecting to the Internet . . . . .	147
3.5	Hub, Switch, Router connecting to the Internet . . . . .	148
3.6	SELinux Context . . . . .	159
3.7	Symmetric Encryption . . . . .	169
3.8	Symmetric Encryption . . . . .	170
3.9	Asymmetric Encryption . . . . .	170
4.1	Example of a process tree . . . . .	183
5.1	Standard Streams . . . . .	218
5.2	File Descriptor Table . . . . .	219
5.3	Pipes . . . . .	232
6.1	Positive and Negative Look-ahead and look-behind assertions . . . . .	292
6.2	Many-to-one mapping . . . . .	295
6.3	Caesar Cipher . . . . .	296
7.1	The <code>comm</code> command . . . . .	319
9.1	Flowchart of the <code>if</code> , <code>elif</code> , and <code>else</code> construct . . . . .	396
10.1	Filtering Streams . . . . .	433
10.2	The different interfaces to <code>sed</code> . . . . .	433
11.1	Alfred Aho . . . . .	461
11.2	Peter Weinberger . . . . .	461

# List of Tables

1.1 Basic Shortcuts in Terminal . . . . .	9
1.2 Basic Commands in GNU Core Utils . . . . .	14
1.3 Manual Page Sections . . . . .	18
1.4 Keys in Info Pages . . . . .	19
1.5 Escape Characters in echo . . . . .	30
1.6 Date Format Specifiers . . . . .	44
1.7 Linux Filesystem Hierarchy . . . . .	47
1.8 Linux Filesystem Directory Classification . . . . .	51
1.9 Octal Representation of Permissions . . . . .	71
1.10 Types of Files . . . . .	73
1.11 Metadata of a File . . . . .	83
1.12 Symlink vs Hard Link . . . . .	89
2.1 History of Vim . . . . .	99
2.2 Ed Commands . . . . .	110
2.3 Commands for location . . . . .	111
2.4 Commands for Editing . . . . .	111
2.5 Ex Commands in Vim . . . . .	124
2.6 Navigation Commands in Vim . . . . .	125
2.7 Moving the Screen Commands in Vim . . . . .	125
2.8 Replacing Text Commands in Vim . . . . .	126
2.9 Toggling Case Commands in Vim . . . . .	126
2.10 Deleting Text Commands in Vim . . . . .	127
2.11 Deleting Text Commands in Vim . . . . .	128
2.12 Address Types in Search and Replace . . . . .	130
2.13 Keys to enter Insert Mode . . . . .	131
2.14 History of Emacs . . . . .	133
2.15 Navigation Commands in Emacs . . . . .	138
2.16 Exiting Emacs Commands . . . . .	138

2.17	Searching Text Commands in Emacs . . . . .	138
2.18	Copying and Pasting Commands in Emacs . . . . .	139
2.19	File Handling Commands in Nano . . . . .	141
2.20	Editing Commands in Nano . . . . .	141
3.1	Private IP Address Ranges . . . . .	154
3.2	Well-known Ports . . . . .	156
3.3	Network Tools . . . . .	160
5.1	Pipes, Streams, and Redirection syntax . . . . .	257
6.1	Differences between BRE and ERE . . . . .	289
9.1	Differences between range expansion and seq command . . . . .	405
9.2	Differences between seq and python's range function . . . . .	406
9.3	Summary of the bash constructs . . . . .	432
11.1	Awk Format Specifiers . . . . .	540

## 1.1 Introduction

### 1.1.1 What is Linux?

**Definition 1.1.1 (Linux)** Linux is a **kernel** that is used in many operating systems. It is open source and free to use. Linux is not an operating system unto itself, but the core component of it.

So what is **Ubuntu**? **Ubuntu** is one of the many *distributions* that use the Linux kernel. It is a complete operating system that is free to use and open source. It is based on the **Debian** distribution of Linux. There are many other *distributions* of Linux, such as:

- ▶ **Debian** - Used primarily on servers, it is known for its stability.
- **Ubuntu** - A commercial distribution based on Debian which is popular among new users.
- **Linux Mint** - A distribution based on Ubuntu which is known for its ease of use. It is one of the distributions recommended to new users.
- **Pop OS** - A distribution based on Ubuntu which is known for its focus on developers, creators, and gamers.
- and many more
- ▶ **Red Hat Enterprise Linux (RHEL)** - A commercial distribution used primarily in enterprises. It is owned by **Red Hat** and is targeted

primarily to companies with their free OS paid support model.

- **Fedora** - A community-driven distribution sponsored by **Red Hat**. It is known for its cutting-edge features and is used by developers. It remains on the upstream of **RHEL**, receiving new features before **RHEL**.
- **CentOS** - A discontinued distribution based on **RHEL**. It was known for its stability and was used in servers. It was downstream from **RHEL**.
- **CentOS Stream** - It is a midstream between the upstream development in **Fedora Linux** and the downstream development for Red Hat Enterprise Linux.
- **Rocky Linux** - A distribution created by the **Rocky Enterprise Software Foundation** after the announcement of discontinuation of **CentOS**. It is a downstream of **RHEL** that provides feature parity and binary compatibility with **RHEL**.
- **Alma Linux** - A distribution created by the **CloudLinux** team after the announcement of discontinuation of **CentOS**. It is a downstream of **RHEL** that provides feature parity and binary compatibility with **RHEL**.
- **Arch Linux** - A community-driven distribution known for its simplicity and customizability. It is a *rolling release* distribution, which means that it is continuously updated. It is a bare-bones distribution that lets the user decide which packages they want to install.
  - **Manjaro** - A distribution based on **Arch Linux** which is known for its user-friendliness. It is a *rolling release* distribution that is

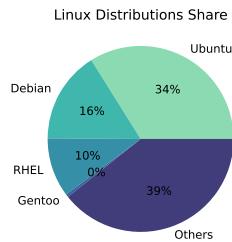
easier to install for new users. It uses a different repository for packages with additional testing.

- **EndeavourOS** - A distribution based on Arch Linux which is known for its simplicity and minimalism. It is a rolling release distribution that is easier to install for new users. It uses the same repository for packages as **Arch Linux**.
- **Artix Linux** - It uses the **OpenRC** init system instead of **systemd**. It also offers other *init systems* like **runit**, **s6**, **dinit**.
- ▶ **openSUSE** - It is a free and open-source Linux distribution developed by the openSUSE project. It is offered in two main variations: **Tumbleweed**, an upstream rolling release distribution, and **Leap**, a stable release distribution which is sourced from SUSE Linux Enterprise.
  - **Tumbleweed** - Rolling Release upstream.
  - **Leap** - Stable Release downstream.
- ▶ **Gentoo** - A distribution known for its customizability and performance. It is a source-based distribution, which means that the user compiles the software from source code. It is known for its performance optimizations for the user's hardware.
- ▶ **Void** - It is an independent rolling-release Linux distribution that uses the X Binary Package System package manager, which was designed and implemented from scratch, and the **runit** init system. Excluding binary kernel blobs, a base install is composed entirely of free<sup>1</sup> software.

1: “Free software” means software that respects users’ freedom and community. Roughly, it means that the users have the freedom to run, copy, distribute, study, change and improve the software. Thus, “free software” is a matter of liberty, not price. To understand the concept, you should think of “free” as in “free speech,” not as in “free beer.” We sometimes call it “libre software,” borrowing the French or Spanish word for “free” as in freedom, to show we do not mean the software is gratis.

You may have paid money to get copies of a free program, or you may have obtained copies at no charge. But regardless of how you got your copies, you always have the freedom to copy and change the software, even to sell copies.

#### - [GNU on Free Software](#)



**Figure 1.1:** Linux Distributions Usage in 2024

## 1.1.2 Desktop Environments

**Definition 1.1.2 (Desktop Environment)** A desktop environment is a collection of software designed to give functionality and a certain look and feel to a desktop operating system. It is a combination of a window manager, a file manager, a panel, and other software that provides a graphical user interface and utilities to a regular desktop user, such as volume and brightness control, multimedia applications, settings panels, etc. This is only required by desktop (and laptop) uses and are not present on server instances.

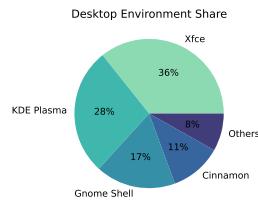
There are many desktop environments available for Linux, but the important ones are:

- 2: GTK is a free software cross-platform widget toolkit for creating graphical user interfaces.
- 3: Ubuntu used to ship with Unity as the default desktop environment, but switched to GNOME in 2017.
- 4: Qt is cross-platform application development framework for creating graphical user interfaces.

- ▶ **GNOME** - One of the most popular desktop environments for Linux. It is known for its simplicity and ease of use. It is the default desktop environment for many distributions, including Ubuntu. It is based on the **GTK Toolkit**.<sup>2</sup> Popular distros shipping by default with GNOME are Fedora, RHEL, CentOS, Debian, Zorin, and Ubuntu.<sup>3</sup>
- ▶ **KDE Plasma** - A highly customizable desktop environment based on the **Qt Toolkit**.<sup>4</sup> Many distributions like Slackware and OpenSUSE ship with KDE Plasma as the default desktop environment, and most others have the option to install with KDE Plasma. Ubuntu's KDE Plasma variant is called **Kubuntu**.
- ▶ **Xfce** - A lightweight desktop environment known for its speed and simplicity. It is based on the **GTK Toolkit**. It is used in many distributions like Xubuntu, Manjaro, and Fedora.

- ▶ **LXQt** - A lightweight desktop environment known for its speed and simplicity. It is based on the **Qt Toolkit**. It is used in many distributions like Lubuntu.
- ▶ **Cinnamon**
- ▶ **MATE**

It is important to note that although some distributions come pre-bundled with certain Desktop Environments, it doesn't mean that you cannot use another DE with it. DE are simply packages installed on your distribution, and almost all the popular DEs can be installed on all distributions. Many distributions also come with multiple pre-bundled desktop environments due to user preferences. Most server distributions and some enthusiast distributions come with no pre-bundled desktop environment, and let the user determine which one is required, or if one is required.



**Figure 1.2:** Desktop Environment Usage in 2022

### 1.1.3 Window Managers

**Definition 1.1.3 (Window Manager)** A window manager is system software that controls the placement and appearance of windows within a windowing system in a graphical user interface. It is a part of the desktop environment, but can also be used standalone. It is responsible for the appearance and placement of windows, and can also provide additional functionality like virtual desktops, window decorations, window title bars, and tiling.

Although usually bundled with a desktop environment, many window managers are also standalone and installed separately by the user if they don't

want to use all the application from a single desktop environment.

Some popular window managers are:

- ▶ **Openbox** - A lightweight window manager known for its speed and simplicity. It is used in many distributions like Lubuntu.
- ▶ **i3** - It is a tiling window manager<sup>5</sup> which is usually one of the first window managers that users try when they want to move away from a desktop environment and to a tiling window manager.
- ▶ **awesome** - A tiling window manager that is highly configurable and extensible. It is written in Lua and is known for its beautiful configurations.
- ▶ **bspwm** - A tiling window manager. It is based on binary space partitioning.
- ▶ **dwm** - A dynamic tiling window manager that is known for its simplicity and minimalism. It is written in C and is highly configurable.

#### 1.1.4 Why Linux?

You might be wondering “*Why should I use Linux?*” Most people use either **Windows** or **Mac** on their personal computers. Although these consumer operating systems get the job done, they don’t let the user completely control their own *hardware* and *software*. Linux<sup>6</sup> is a free and open-source operating system that gives the user complete control over their system. It is highly customizable and can be tailored to the user’s needs. It is also known for its stability and security. It is used in almost all servers, supercomputers, and embedded systems. It

5: A tiling window manager is a window manager that automatically splits the screen into non-overlapping frames, which are used to display windows. Most desktop environments ship with a **floating** window manager instead, which users of other operating systems are more familiar with.

6: Although Linux is just a kernel and not an entire operating system, throughout this book I would be referring to GNU/Linux, the combination of **GNU core utilities** and the Linux kernel, as **Linux** in short.

is also used in many consumer devices like Android phones, smart TVs, and smartwatches.

In this course we will be covering how to navigate the linux file system, how to manage files, how to manage the system, and how to write scripts to automate tasks. In the later part of the course we go over concepts such as pattern matching and text processing.

This course does not go into details of the linux kernel, but rather attempts to make the reader familiar with the *GNU core utils* and able to navigate around a linux server easily.

### 1.1.5 What is Shell?

The **kernel** is the core of the operating system. It is responsible for managing the hardware and providing services to the user programs. The **shell** is the interface between the user and the kernel (Figure 1.3). Through the **shell** we can run many commands and utilities, as well as some inbuilt features of the shell.

**Definition 1.1.4 (Shell)** A shell is a command-line interpreter that provides a way for the user to interact with the operating system. It takes commands from the user and executes them. It is a program that provides the user with a way to interact with the operating system.

The most popular shell in Linux is the **bash** shell. It is the default shell in most distributions. It is a POSIX-compliant<sup>7</sup> shell. There are many other shells available, such as **zsh**, **fish**<sup>8</sup>, **dash**, **csh**, **ksh**, and **tcsh**. Each shell has its own features and syntax,

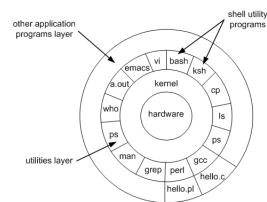


Figure 1.3: Operating System Onion Rings - The layers of an operating system

7: POSIX, or Portable Operating System Interface, is a set of standards that define the interfaces and environment that operating systems use to access POSIX-compliant applications. POSIX standards are based on the Unix operating system and were released in the late 1980s.

8: Fish is a non-POSIX compliant shell that is known for its features like auto-suggestion, syntax highlighting, and tab completions. Although a useful alternative to other shells for scripting, it should not be set as the default shell.

but most of the keywords and syntax are the same. In this course we will be covering only the **bash** shell and its syntax, but most of what we learn here is also applicable on other shells as well.

### 1.1.6 Shell vs Terminal

**Definition 1.1.5** (Terminal) A terminal is a program that provides a way to interact with the shell. It is a program that provides a text-based interface to the shell. It is also known as a terminal emulator.

The terminal is the window that you see when you open a terminal program. It provides a way to interact with the shell. The shell is the program that interprets the commands that you type in the terminal. The terminal is the window that you see, and the shell is the program that runs in that window. Whereas the shell is the application that is parsing your input and running the commands and keywords, the terminal is the application that lets you see the shell graphically. There are multiple different terminal emulators, providing a lot of customization and beautification to the terminal, as well as providing useful features such as *scroll back*, copying and pasting, and so on.

Some popular terminal emulators are:

- ▶ **gnome-terminal** - The default terminal emulator for the GNOME desktop environment.
- ▶ **konsole** - The default terminal emulator for the KDE desktop environment.
- ▶ **xfce4-terminal** - The default terminal emulator for the Xfce desktop environment.
- ▶ **alacritty** - A terminal emulator known for its speed and simplicity.

- ▶ **terminator** - A terminal emulator known for its features like splitting the terminal into multiple panes.
- ▶ **tilix** - A terminal emulator known for its features like splitting the terminal into multiple panes.
- ▶ **st** - A simple terminal emulator known for its simplicity and minimalism.
- ▶ **urxvt**
- ▶ **kitty**
- ▶ **terminology**

In most terminal emulators, there are some basic shortcuts that can be used to make the terminal experience more efficient. Some of the basic shortcuts are listed in Table Table 1.1.

**Table 1.1:** Basic Shortcuts in Terminal

Shortcut	Description
Ctrl + C	Terminate the current process
Ctrl + D	Exit the shell
Ctrl + L	Clear the terminal screen
Ctrl + A	Move the cursor to the beginning of the line
Ctrl + E	Move the cursor to the end of the line
Ctrl + U	Delete from the cursor to the beginning of the line
Ctrl + K	Delete from the cursor to the end of the line
Ctrl + W	Delete the word before the cursor
Ctrl + Y	Paste the last deleted text
Ctrl + R	Search the command history
Ctrl + Z	Suspend the current process
Ctrl + \\	Terminate the current process
Ctrl + S	Pause the terminal output
Ctrl + Q	Resume the terminal output

### 1.1.7 Why the Command Line?

Both the command line interface (CLI) and the graphical user interface (GUI) are simply shells over the operating system's kernel. They let you interact with the kernel, perform actions and run applications.

#### **GUI:**

The GUI requires a mouse and a keyboard, and is more intuitive and easier to use for beginners. But it is also slower and less efficient than the CLI. The GUI severely limits the user's ability to automate tasks and perform complex operations. The user can only perform those operations that the developers of the GUI have thought of and implemented.

#### **CLI:**

The CLI is faster and more efficient than the GUI as it lets the user use the keyboard to perform actions. Instead of clicking on pre-defined buttons, the CLI lets you construct your instruction to the computer using syntax and semantics. The CLI lets you combine simple commands that do one thing well to perform complex operations. The biggest advantage of the CLI is that it lets you automate tasks. It might be faster for some users to rename a file from **file1** to **file01** using the GUI, but it will always be faster to automate this using the CLI if you want to do this for thousands of files in the folder.

In this course we will be learning how to use the CLI to navigate the file system, manage files, manage the system, process text, and write scripts to automate tasks.

### 1.1.8 Command Prompt

The command prompt is the text that is displayed in the terminal to indicate that the shell is ready to accept commands. It usually ends with a \$ or a # symbol. The \$ symbol indicates that the shell is running as a normal user, and the # symbol indicates that the shell is running as the root user. The root user has complete control over the system and can perform any operation on the system.

An example of a command prompt is:

```
1 | username@hostname:~$
```

Here, `username` is the name of the user, `hostname` is the name of the computer, and \$ indicates that the shell is running as a normal user. The ~ symbol indicates that the current working directory is the user's home directory. The home directory is the directory where the user's files and settings are stored. It is usually located at `/home/username`. This can be shortened to ~ in the shell. This prompt can be changed and customized according to the user's preferences using the `PS1` variable discussed in Chapter 8.

## 1.2 Simple Commands in GNU Core Utils



**Figure 1.4:** GNU Core Utils Logo

**Definition 1.2.1 (GNU Core Utils)** The GNU Core Utilities are the basic file, shell, and text manipulation utilities of the GNU operating system. These are the utilities that are used to interact with the operating system and perform basic operations.<sup>a</sup>

<sup>a</sup> [GNU Core Utils](#)

The shell lets you simply type in the name of the command and press enter to run it. You can also pass arguments to the command to modify its behavior. Although the commands are simple, they are powerful and can be combined to perform complex operations.<sup>9</sup>

Some basic commands in the core-utils are listed in Table Table 1.2 on page 14.

9: The combination of commands to perform complex operations is called *piping*. This will be covered later.

### 1.2.1 File System Navigation

**pwd:**

The `pwd` command prints the current working directory. The current working directory is the directory that the shell is currently in. The shell starts in the user's home directory when it is opened. The `pwd` command prints the full path of the current working directory.

```
1 | $ pwd
2 | /home/username
```

**ls:**

The `ls` command lists the contents of a directory. By default, it lists the contents of the current working directory. The `ls` command can take arguments to list the contents of a different directory.

```
1 $ ls  
2 Desktop Documents Downloads Music  
    Pictures Videos
```

We can also list hidden files<sup>10</sup> using the `-a` flag.

```
1 $ ls -a  
2 . .. .bashrc Desktop Documents  
    Downloads Music Pictures Videos
```

10: Hidden files are files whose names start with a dot. They are hidden by default in the `ls` command.

Here, the `.` and `..` directories are special directories. The `.` directory is the current directory, and the `..` directory is the parent directory. The `.bashrc` file is a configuration file for the shell which is a hidden file.

`ls` can also list the details of the files using the `-l` flag.

```
1 $ ls -l  
2 total 24  
3 drwxr-xr-x 2 username group 4096 Mar  1  
    12:00 Desktop  
4 drwxr-xr-x 2 username group 4096 Mar  1  
    12:00 Documents  
5 drwxr-xr-x 2 username group 4096 Mar  1  
    12:00 Downloads  
6 drwxr-xr-x 2 username group 4096 Mar  1  
    12:00 Music  
7 drwxr-xr-x 2 username group 4096 Mar  1  
    12:00 Pictures  
8 drwxr-xr-x 2 username group 4096 Mar  1  
    12:00 Videos
```

**Table 1.2:** Basic Commands in GNU Core Utils

Command	Description
<code>ls</code>	List the contents of a directory
<code>cd</code>	Change the current working directory
<code>pwd</code>	Print the current working directory
<code>mkdir</code>	Create a new directory
<code>rmdir</code>	Remove a directory
<code>touch</code>	Create a new file
<code>rm</code>	Remove a file
<code>cp</code>	Copy a file
<code>mv</code>	Move a file
<code>echo</code>	Print a message
<code>cat</code>	Concatenate and display the contents of a file
<code>less</code>	Display the contents of a file one page at a time
<code>head</code>	Display the first few lines of a file
<code>tail</code>	Display the last few lines of a file
<code>find</code>	Find files and directories
<code>locate</code>	Find files and directories
<code>which</code>	Find the location of a command
<code>uname</code>	Print system information
<code>ps</code>	Display information about running processes
<code>kill</code>	Terminate a process
<code>chmod</code>	Change the permissions of a file
<code>chown</code>	Change the owner of a file
<code>chgrp</code>	Change the group of a file
<code>date</code>	Print the current date and time
<code>cal, ncal</code>	Print a calendar
<code>df</code>	Display disk space usage
<code>du</code>	Display disk usage
<code>free</code>	Display memory usage
<code>top</code>	Display system information
<code>history</code>	Display the command history
<code>sleep</code>	Pause the shell for a specified time
<code>true</code>	Do nothing, successfully
<code>false</code>	Do nothing, unsuccessfully
<code>tee</code>	Read from stdin and write to stdout and files
<code>whoami</code>	Print the current user
<code>groups</code>	Print the groups the user belongs to
<code>clear</code>	Clear the terminal screen
<code>exit</code>	Exit the shell

As seen in Figure 1.5, the first column is the file type and permissions. The second column is the number of links to the file or directory. The third and fourth columns are the owner and group of the file or directory. The fifth column is the size of the file or directory. The sixth, seventh, and eighth columns are the last modified date and time of the file or directory. The ninth column is the name of the file or directory.<sup>11</sup>

We can also list the inode numbers<sup>12</sup> using the `-i` flag.

```
1 $ ls -i
2 123456 Desktop 123457 Documents 123458
   Downloads 123459 Music 123460 Pictures
   123461 Videos
```

Inodes will be discussed in detail later in the course.

**cd:**

The `cd` command changes the current working directory. It takes the path to the directory as an argument.

```
1 $ cd Documents
2 $ pwd
3 /home/username/Documents
```

The `cd` command can also take the `~` symbol as an argument to change to the user's home directory. This is the default behavior of the `cd` command when no arguments are passed.

```
1 $ cd
2 $ pwd
3 /home/username
```

If we want to go back to the previous directory, we can use the `-` symbol as an argument to the `cd` command.<sup>13</sup>



Figure 1.5: `ls -l` Output

11: More details about the file permissions and the file types will be covered later in the course.

12: An inode is a data structure on a filesystem on Linux and other Unix-like operating systems that stores all the information about a file except its name and its actual data. This includes the file type, the file's owner, the file's group, the file's permissions, the file's size, the file's last modified date and time, and the file's location on the disk. The inode is the reference pointer to the data in the disk.

13: This internally uses the `OLDPWD` environment variable to change the directory. More about variables will be covered later in the course.

```

1 | $ cd Documents
2 | $ pwd
3 | /home/username/Documents
4 | $ cd -
5 | $ pwd
6 | /home/username

```

14: cwd means Current Working Directory

**Question 1.2.1** `ls` can only show files and directories in the cwd<sup>14</sup>, not subdirectories. True or False?

**Answer 1.2.1** False. `ls` can show files and directories in the cwd, and also in subdirectories. The `-R` flag can be used to show files and directories in subdirectories, recursively.

## 1.2.2 Manuals

**man:**

How to remember so many flags and options for each of the commands? The `man` command is used to display the manual pages for a command.

**Definition 1.2.2 (Manual Pages)** Manual pages are a type of software documentation that provides details about a command, utility, function, or file format. They are usually written in a simple and concise manner and provide information about the command's syntax, options, and usage.

```

1 | $ man ls

```

This will display the manual page for the `ls` command. The manual page is divided into sections,

and you can navigate through the sections using the arrow keys. Press q to exit the manual page.

Example manual page:

```
1 LS(1)
2
3           User Commands
4
5           LS(1)
6
7 NAME
8       ls - list directory contents
9
10 SYNOPSIS
11      ls [OPTION]... [FILE]...
12
13 DESCRIPTION
14      List information about the FILEs (the
15      current directory by default). Sort
16      entries alphabetically if none of -
17      cftuvSUX nor --sort is specified.
18
19      Mandatory arguments to long options are
20      mandatory for short options too.
21
22      -a, --all
23          do not ignore entries starting
24          with .
25
26      -A, --almost-all
27          do not list implied . and ..
28 ...
29
```

The manual page provides information about the command, its syntax, options, and usage. It is a good practice to refer to the manual page of a command before using it.

To exit the manual page, press q.

There are multiple sections in the manual page, `man` takes the section number as an argument to display the manual page from that section.

```
1| $ man 1 ls
```

This will display the manual page for the `ls` command from section 1. The details of the sections can be seen in Table Table 1.3.

**Table 1.3:** Manual Page Sections

Section	Description
1	User Commands
2	System Calls
3	Library Functions
4	Special Files usually found in /dev
5	File Formats and conventions
6	Games
7	Miscellaneous
8	System Administration
9	Kernel Developer's Manual

Man pages only provide information about the commands and utilities that are installed on the system. They do not provide information about the shell builtins or the shell syntax. For that, you can refer to the shell's documentation or use the `help` command.

Some commands also have a `--help` flag that displays the usage and options of the command.

Some commands have their own `info` pages, which are more detailed than the `man` pages.

To be proficient with shell commands, one needs to read the `man`, `info`, and `help` pages.<sup>15</sup>

15: An useful video by Robert Elder about the differences between `man`, `info`, and `help` can be found on YouTube.

**Exercise 1.2.1** Run `man`, `info`, and `--help` on all the commands discussed in this section. Note

the differences in the information provided by each of them. Read the documentations carefully and try to understand how each command works, and the pattern in which the documentations are written.

### **info:**

The **info** command is used to display the info pages for a command. The info pages are more detailed than the **man** pages for some commands. It is navigable like a hypertext document. There are links to chapters inside the info pages that can be followed using the arrow keys and entered using the enter key. The Table Table 1.4 lists some of the keys that can be used to navigate the info pages.

Key	Description
h	Display the help page
q	Exit the info page
n	Move to the next node
p	Move to the previous node
u	Move up one level
d	Move down one level
l	Move to the last node
t	Move to the top node
g	Move to a specific node
<enter>	Follow the link
m	Display the menu
s	Search for a string
S	Search for a string (case-sensitive)

**Table 1.4:** Keys in Info Pages

### **help:**

The **help** command is a shell builtin that displays information about the shell builtins and the shell syntax.

1 | \$ **help read**

This will list the information about the `read` builtin command.

The `help` command can also be used to display information about the shell syntax.

```
1 | $ help for
```

This will list the information about the `for` loop in the shell.

Help pages are not paged, and the output is displayed in the terminal. To page the output, one can use the `less` command.

```
1 | $ help read | less
```

### 1.2.3 System Information

#### `uname`:

The `uname` command prints system information. It can take flags to print specific information about the system. By default, it prints only the kernel name.

```
1 | $ uname  
2 | Linux
```

16: Here `rex` is the host-name of the system, `6.8.2-arch2-1` is the kernel version, `x86_64` is the architecture, and `GNU/Linux` is the operating system.

The `-a` flag prints all the system information.<sup>16</sup>

```
$ uname -a  
Linux rex 6.8.2-arch2-1 #1 SMP  
PREEMPT_DYNAMIC Thu, 28 Mar 2024 17:06:35  
+0000 x86_64 GNU/Linux
```

#### `ps`:

The `ps` command displays information about running processes. By default, it displays information about the processes run by the current user that are running from a terminal.<sup>17</sup>

17: The **PID** is the process ID, the **TTY** is the terminal the process is running from, the **TIME** is the time the process has been running, and the **CMD** is the command that is running.

```

1 $ ps
2   PID TTY      TIME CMD
3 12345 pts/0    00:00:00 bash
4 12346 pts/0    00:00:00 ps

```

There are a lot of flags that can be passed to the `ps` command to display more information about the processes. These will be covered in Chapter 4.

#### Remark 1.2.1 `ps` has three types of options:

- ▶ UNIX options
- ▶ BSD options
- ▶ GNU options

The UNIX options are the preceded by a hyphen (-) and may be grouped. The BSD options can be grouped, but should not be preceded by a hyphen (-). The GNU options are preceded by two hyphens (--). These are also called long options.

The same action can be performed by using different options, for example, `ps -ef` and `ps aux` are equivalent, although first is using **UNIX** options, and the latter is using **BSD** options.

Another difference in **GNU core utils** and **BSD utils** is that the **GNU** utils have long options, whereas the **BSD** utils do not.

**BSD** utils also usually do not support having flags after the positional arguments, whereas most **GNU** utils are fine with this.

#### kill:

The `kill` command is used to terminate a process. It takes the process ID as an argument.

```

1 $ kill 12345

```

18: The SIGKILL signal is used to terminate a process immediately. It cannot be caught or ignored by the process. It is numbered as 9.

The `kill` command can also take the **signal number** as an argument to send a signal to the process. For example, the **SIGKILL** signal can be sent to the process to terminate it.<sup>18</sup>

```
1 | $ kill -9 12345
```

### **free:**

The `free` command is used to display the amount of free and used memory in the system.

```
1 | $ free
2 |          total        used        free
3 |          shared    buff/cache   available
4 | Mem:      8167840      1234560     4567890
4 |           123456      2367890     4567890
4 | Swap:     2097148          0     2097148
```

The `free` command can take the `-h` flag to display the memory in human-readable format.

```
1 | $ free -h
2 |          total        used        free
3 |          shared    buff/cache   available
4 | Mem:      7.8Gi       1.2Gi       4.3Gi
4 |       120Mi       2.3Gi       4.3Gi
4 | Swap:     2.0Gi          0B       2.0Gi
```

### **df:**

The `df` command is used to display the amount of disk space available on the filesystems.

```
1 | $ df
2 | Filesystem  1K-blocks  Used Available
3 |          Use% Mounted on
4 | /dev/sda1      12345678 1234567  11111111
4 |          10% /
4 | /dev/sda2      12345678 1234567  11111111
4 |          10% /home
```

The `df` command can take the `-h` flag to display the disk space in human-readable format.

```

1 $ df -h
2 Filesystem      Size  Used Avail Use%
3           Mounted on
4 /dev/sda1        12G  1.2G  9.9G  11% /
4 /dev/sda2        12G  1.2G  9.9G  11% /home

```

**du:**

The **du** command is used to display the disk usage of directories and files. By default, it displays the disk usage of the current directory.

```

1 $ du
2 4     ./Desktop
3 4     ./Documents
4 4     ./Downloads
5 4     ./Music
6 4     ./Pictures
7 4     ./Videos
8 28

```

The **du** command can take the **-h** flag to display the disk usage in human-readable format. The **-s** flag displays the total disk usage of the directory.

```

1 $ du -sh
2 28K   .

```

**Question 1.2.2** How to print the kernel version of your system?

**Answer 1.2.2** `uname -r` will print the kernel version of your system. `uname` is a command to print system information. The `-r` flag is to print the kernel release. There are other flags to print other system information.

We can also run `uname -a` to get all fields and extract only the kernel info using commands taught in later weeks.

**Question 1.2.3** How to see how long your system is running for?

What about the time it was booted up?

**Answer 1.2.3** `uptime` will show how long the system is running for.

`uptime -s` will show the time the system was booted up.

The `-s` flag is to show the time of last boot.

**Question 1.2.4** How to see the amount of free memory? What about free hard disk space? If we are unable to understand the big numbers, how to convert them to human readable format?  
What is difference between MB and MiB?

**Answer 1.2.4** `free` will show the amount of free memory.

`df` will show the amount of free hard disk space.  
`df -h` and `free -h` will convert the numbers to human readable format.

MB is Megabyte, and MiB is Mebibyte.

1 MB = 1000 KB, 1 GB = 1000 MB, 1 TB = 1000 GB, this is SI unit.

1 MiB = 1024 KiB, 1 GiB = 1024 MiB, 1 TiB = 1024 GiB, this is  $2^{10}$  unit.

## 1.2.4 File Management

**file:**

The `file` command is used to determine the type of a file. It can take multiple file names as arguments.

```

1 $ file file1
2 file1: ASCII text
3 $ file /bin/bash
4 /bin/bash: ELF 64-bit LSB shared object, x86
   -64, version 1 (SYSV), dynamically linked,
   interpreter /lib64/ld-linux-x86-64.so.2,
   for GNU/Linux 3.2.0, BuildID[sha1]
   ]=1234567890abcdef, stripped

```

### **mkdir:**

The `mkdir` command is used to create new directories. It can take multiple directory names as arguments.

```

1 $ mkdir a b c
2 $ ls -F
3 a/ b/ c/

```

**Exercise 1.2.2** Run `man ls` to find out what the `-F` flag does, and why we used it in the above example.

### **touch:**

The `touch` command is used to create new files. It can take multiple file names as arguments. If a file is already present, the `touch` command updates the last modified date and time of the file, but does not modify the contents of the file.

```

1 $ touch file1 file2 file3
2 $ ls -l
3 -rw-r--r-- 1 username group 0 Mar  1 12:00
   file1
4 -rw-r--r-- 1 username group 0 Mar  1 12:00
   file2
5 -rw-r--r-- 1 username group 0 Mar  1 12:00
   file3
6 $ sleep 60
7 $ touch file3

```

```

8 | $ ls -l
9 | -rw-r--r-- 1 username group 0 Mar  1 12:00
   |     file1
10 | -rw-r--r-- 1 username group 0 Mar  1 12:00
    |     file2
11 | -rw-r--r-- 1 username group 0 Mar  1 12:01
    |     file3

```

**Exercise 1.2.3** Notice the difference in the last modified date and time of the `file3` file from the other files. Also notice the `sleep` command used to pause the shell for 60 seconds.

### **rmdir:**

The `rmdir` command is used to remove directories. It can take multiple directory names as arguments.

```

1 | $ mkdir a b c d
2 | $ rmadir a b c
3 | $ ls -F
4 | d/

```

**Remark 1.2.2** The `rmadir` command can only remove empty directories. This is a safety feature so that users don't accidentally delete directories with files in them. To remove directories with files in them along with those files, use the `rm` command.

### **rm:**

The `rm` command is used to remove files and directories. It can take multiple file and directory names as arguments.

```

1 | $ touch file1 file2 file3
2 | $ ls -F
3 | file1  file2  file3

```

```

4 | $ rm file1 file2
5 | $ ls -F
6 | file3

```

However, using `rm` to delete a directory will give an error.

```

1 | $ mkdir a
2 | $ rm a
3 | rm: cannot remove 'a': Is a directory

```

This is because the `rm` command does not remove directories by default. This is a safety feature to prevent users from accidentally deleting directories with files in them.

To remove directories along with their files, use the `-r` flag.

```
1 | $ rm -r a
```

To force the removal of files and directories without a confirmation, use the `-f` flag.

**Warning 1.2.1** The `rm` command is a dangerous command. It does not move the files to the trash, but permanently deletes them. Be **extremely** careful when using the `rm` command. Only use the `-f` flag if you are absolutely sure that you want to delete the files.

To force `rm` to always ask for confirmation before deleting files, use the `-i` flag.

```

1 | $ rm -i file3
2 | rm: remove regular empty file 'file3'? y

```

**cp:**

The `cp` command is used to copy files. It takes the source file and the destination file as arguments.

```

1 | $ touch file1
2 | $ ls -F
3 | file1
4 | $ cp file1 file2
5 | $ ls -F
6 | file1  file2

```

The `cp` command can also take the `-r` flag to copy directories.

```

1 | $ mkdir a
2 | $ touch a/file1
3 | $ cp -r a b
4 | $ ls -R
5 | .:
6 | a/  b/
7 |
8 | ./a:
9 | file1
10|
11| ./b:
12| file1

```

**Exercise 1.2.4** Why did we use the `-R` flag in the above example? What does it do?

There are three ways to copy files using `cp`:

```

1 | SYNOPSIS
2 |     cp [OPTION]... [-T] SOURCE DEST
3 |     cp [OPTION]... SOURCE... DIRECTORY
4 |     cp [OPTION]... -t DIRECTORY SOURCE...

```

**Exercise 1.2.5** There are three ways of running the `cp` command to copy a file. Here we have demonstrated only one. Read the manual page of the `cp` command to find out the other two ways and try them out yourself.

**mv:**

The `mv` command is used to move files. The syntax is similar to the `cp` command.<sup>19</sup> It is used to move files from one location to another, or to rename files.

19: This means that `mv` also has three ways of running it.

```
1 $ touch file1
2 $ ls -F
3   file1
4 $ mv file1 file2
5 $ ls -F
6   file2
```

**Exercise 1.2.6** Create a directory `dir1` using the `mkdir` command, then create a file `file1` inside `dir1`. Now move (rename) the `dir1` directory to `dir2` using the `mv` command. The newly created directory should be named `dir2` and should contain the `file1` file. Were you require to use the `-r` flag with the `mv` command like you would have in `cp` command?

## 1.2.5 Text Processing and Pagers

**echo:**

The `echo` command is used to print a message to the terminal. It can take multiple arguments and print them to the terminal.

```
1 $ echo Hello, World!
2   Hello, World!
```

The `echo` command can also take the `-e` flag to interpret backslash escapes.

```
1 $ echo -e "Hello, \nWorld!"
2   Hello,
3   World!
```

Some escape characters in **echo** are listed in Table Table 1.5.

**Table 1.5:** Escape Characters in echo

Escape	Description
\\\	backslash
\a	alert (BEL)
\b	backspace
\c	produce no further output
\e	escape
\f	form feed
\n	new line
\r	carriage return
\t	horizontal tab
\v	vertical tab
\0NNN	byte with octal value NNN (1 to 3 digits)
\xHH	byte with hexadecimal value HH (1 to 2 digits)

**Exercise 1.2.7** Run the command `echo -e "\x41=\x0101"` and try to understand the output and the escape characters used.

### cat:

The **cat** command is used to concatenate and display the contents of files.

```
1 | $ cat file1
2 | Hello, World! from file1
```

The **cat** command can take multiple files as arguments and display their contents one after another.

```
1 | $ cat file1 file2
2 | Hello, World! from file1
3 | Hello, World! from file2
```

### less:

Sometimes the contents of a file are too large to be displayed at once. Nowadays modern terminal emulators can scroll up and down to view the contents of the file, but actual ttys cannot do that. To view the contents of a file one page at a time, use the less command. less is a pager program that displays the contents of a file one page at a time.

<sup>20</sup>

```
1 | $ less file1
```

To scroll up and down, use the arrow keys, or the j and k keys.<sup>21</sup> Press q to exit the less command.

### head:

The head command is used to display the first few lines of a file. By default, it displays the first 10 lines of a file.

```
1 | $ head /etc/passwd
2 | root:x:0:0:root:/root:/usr/bin/bash
3 | bin:x:1:1:::/usr/bin/nologin
4 | daemon:x:2:2:::/usr/bin/nologin
5 | mail:x:8:12::/var/spool/mail:/usr/bin/
   nologin
6 | ftp:x:14:11::/srv/ftp:/usr/bin/nologin
7 | http:x:33:33::/srv/http:/usr/bin/nologin
8 | nobody:x:65534:65534:Kernel Overflow User
   ::/usr/bin/nologin
9 | dbus:x:81:81:System Message Bus:::/usr/bin/
   nologin
10 | systemd-coredump:x:984:984:systemd Core
    Dumper:::/usr/bin/nologin
11 | systemd-network:x:982:982:systemd Network
   Management:::/usr/bin/nologin
```

The head command can take the -n flag to display the first *n* lines of a file.<sup>22</sup>

```
1 | $ head -n 5 /etc/passwd
2 | root:x:0:0:root:/root:/usr/bin/bash
3 | bin:x:1:1:::/usr/bin/nologin
```

<sup>20</sup>: more is another pager program that displays the contents of a file one page at a time. It is older than less and has fewer features. less is an improved version of more and is more commonly used. Due to this, it is colloquially said that "less is more", as it has more features.

<sup>21</sup>: Using j and k to move the cursor up and down is a common keybinding in many terminal applications. This originates from the vi text editor which will be covered later in the course.

<sup>22</sup>: We can also directly run head -5 /etc/passwd to display the first 5 lines of the file.

```

4  daemon:x:2:2:::/usr/bin/nologin
5  mail:x:8:12::var/spool/mail:/usr/bin/
   nologin
6  ftp:x:14:11::srv/ftp:/usr/bin/nologin

```

**Remark 1.2.3** Here we are listing the file /etc/passwd which contains information about the users on the system. The file will usually be present on all Unix-like systems and have a lot of system users.<sup>23</sup>

23: A system user is a user that is used by the system to run services and daemons. It does not belong to any human and usually is not logged into. System users have a user ID less than 1000.

### tail:

The tail command is used to display the last few lines of a file. By default, it displays the last 10 lines of a file.

```

1  $ tail /etc/passwd
2  rtkit:x:133:133:RealtimeKit:/proc:/usr/bin/
   nologin
3  sddm:x:964:964:SDDM Greeter Account:/var/lib/
   /sddm:/usr/bin/nologin
4  usbmux:x:140:140:usbmux user:/:/usr/bin/
   nologin
5  sayan:x:1000:1001:Sayan:/home/sayan:/bin/
   bash
6  qemu:x:962:962:QEMU user:/:/usr/bin/nologin
7  cups:x:209:209:cups helper user:/:/usr/bin/
   nologin
8  dhcpcd:x:959:959:dhcpcd privilege separation
   ::/usr/bin/nologin
9  saned:x:957:957:SANE daemon user:/:/usr/bin/
   nologin

```

The tail command can take the -n flag to display the last *n* lines of a file.

```

1  $ tail -n 5 /etc/passwd
2  sayan:x:1000:1001:Sayan:/home/sayan:/bin/
   bash
3  qemu:x:962:962:QEMU user:/:/usr/bin/nologin

```

```
4 cups:x:209:209:cups helper user:::/usr/bin/  
    nologin  
5 dhcpcd:x:959:959:dhcpcd privilege separation  
    ::/usr/bin/nologin  
6 saned:x:957:957:SANE daemon user:::/usr/bin/  
    nologin
```

**Exercise 1.2.8** Notice that the UID (3rd column) of the sayan user is 1000. The last column is /bin/bash instead of /usr/bin/nologin like others. This is because it is a regular user and not a system user.

### wc:

The wc command is used to count the number of lines, words, and characters in a file. By default, it displays the number of lines, words, and characters in a file.

```
1 $ wc /etc/passwd  
2   43 103 2426 /etc/passwd
```

We can also use the -l, -w, and -c flags to display only the number of lines, words, and characters respectively.

```
1 $ wc -l /etc/passwd  
2 43 /etc/passwd
```

**Question 1.2.5** Can we print contents of multiple files using a single command?

**Answer 1.2.5** cat file1 file2 file3 will print the contents of file0, file2, and file3 in the order given. The contents of the files will be printed one after the other.

**Question 1.2.6** Can cat also be used to write to a file?

**Answer 1.2.6** Yes, `cat > file1` will write to `file1`. The input will be taken from the terminal and written to `file1`. The input will be written to `file1` until the user presses `Ctrl+D` to indicate end of input. This is redirection, which we see in later weeks.

**Question 1.2.7** How to list only first 10 lines of a file? How about first 5? Last 5? How about lines 105 to lines 152?

**Answer 1.2.7** `head filename` will list the first 10 lines of `filename`.

`head -n 5 filename` will list the first 5 lines of `filename`.

`tail -n 5 filename` will list the last 5 lines of `filename`.

`head -n 152 filename tail -n 48 |` will list lines 105 to 152 of `filename`. This uses `|` which is a pipe, which we will see in later weeks.

**Question 1.2.8** Do you know how many lines a file contains? How can we count it? What about words? Characters?

**Answer 1.2.8** `wc filename` will count the number of lines, words, and characters in `filename`. `wc -l filename` will count the number of lines in `filename`.

`wc -w filename` will count the number of words

in filename.

`wc -c filename` will count the number of characters in filename.

**Question 1.2.9** How to delete an empty directory? What about non-empty directory?

**Answer 1.2.9** `rmdir dirname` will delete an empty directory.

`rm -r dirname` will delete a non-empty directory.

**Question 1.2.10** How to copy an entire folder to another name? What about moving?

Why the difference in flags?

**Answer 1.2.10** `cp -r sourcefolder targetfolder` will copy an entire folder to another name.

`mv sourcefolder targetfolder` will move an entire folder to another name.

The difference in flags is because `cp` is used to copy, and `mv` is used to move or rename a file or folder. The `-r` flag is to copy recursively, and is not needed for `mv` as it is not recursive and simply changes the name of the folder (or the path).

## 1.2.6 Aliases and Types of Commands

**alias:**

The `alias` command is used to create an alias for a command. An alias is a custom name given to a

24: Aliases are used to create shortcuts for long commands. They can also be used to create custom commands.

command that can be used to run the command.

24

```
1 $ alias ll='ls -l'
2 $ ll
3   total 24
4   drwxr-xr-x 2 username group 4096 Mar  1
5       12:00 Desktop
6   drwxr-xr-x 2 username group 4096 Mar  1
7       12:00 Documents
8   drwxr-xr-x 2 username group 4096 Mar  1
9       12:00 Downloads
10  drwxr-xr-x 2 username group 4096 Mar  1
11      12:00 Music
12  drwxr-xr-x 2 username group 4096 Mar  1
13      12:00 Pictures
14  drwxr-xr-x 2 username group 4096 Mar  1
15      12:00 Videos
```

The alias `ll` is created for the `ls -l` command.

**Warning 1.2.2** Be careful when creating aliases. Do not create aliases for existing commands. This can lead to confusion and errors.

But this alias is temporary and will be lost when the shell is closed. To make the alias permanent, add it to the shell configuration file. For `bash`, this is the `.bashrc` file in the home directory.

```
1 $ echo "alias ll='ls -l'" >> ~/.bashrc
```

This will add the alias to the `.bashrc` file. To make the changes take effect, either close the terminal and open a new one, or run the `source` command.

```
1 $ source ~/.bashrc
```

**Warning 1.2.3** Be careful when editing the shell configuration files. A small mistake can lead to

the shell not working properly. Always keep a backup of the configuration files before editing them.

We can see the aliases that are currently set using the `alias` command.

```
1 | $ alias
2 | alias ll='ls -l'
```

We can also see what a particular alias expands to using the `alias` command with the alias name as an argument.

```
1 | $ alias ll
2 | alias ll='ls -l'
```

To remove an alias, use the `unalias` command.

```
1 | $ unalias ll
```

**Exercise 1.2.9** Create an alias `la` for the `ls -a` command. Make it permanent by adding it to the `.bashrc` file. Check if the alias is set using the `alias` command.

**Exercise 1.2.10** Create an alias `rm` for the `rm -i` command. Make it permanent by adding it to the `.bashrc` file. Check if the alias is set using the `alias` command. Try to delete a file using the `rm` command. What happens?

**which:**

The `which` command is used to show the path of the command that will be executed.

```
1 | $ which ls
2 | /sbin/ls
```

We can also list all the paths where the command is present using the `-a` flag.

```
1 $ which -a ls
2 /sbin/ls
3 /bin/ls
4 /usr/bin/ls
5 /usr/sbin/ls
```

This means that if we delete the `/sbin/ls` file, the `/bin/ls` file will be executed when we run the `ls` command.

#### **whatis:**

The `whatis` command is used to show a short description of the command.

```
1 $ whatis ls
2 ls (1)           - list directory contents
3 ls (1p)          - list directory contents
```

Here the brackets show the section of the manual where the command is present. This short excerpt is taken from its man page itself.

#### **whereis:**

The `whereis` command is used to show the location of the command, source files, and man pages.

```
1 $ whereis ls
2 ls: /usr/bin/ls /usr/share/man/man1/ls.1.gz /
      usr/share/man/man1p/ls.1p.gz
```

Here we can see that the `ls` command is present in `/usr/bin/ls`, and its man pages are present in `/usr/share/man/man1/ls.1.gz` and `/usr/share/man/man1p/ls.1p.gz`.

#### **locate:**

The `locate` command is used to find files by name. The file can be present anywhere in the system and

if it is indexed by the `mlocate` database, it can be found using the `locate` command.

```
1 $ touch tmp/48/hellohowareyou
2 $ pwd
3 /home/sayan
4 $ locate hellohowareyou
5 /home/sayan/tmp/48/hellohowareyou
```

Note: you may have to run `updatedb` to update the database before using `locate`. This can only be run by the root user or using `sudo`.

#### **type:**

The `type` command is used to show how the shell will interpret a command. Usually some commands are both an executable and a shell built-in. The `type` command will show which one will be executed.

```
1 $ type ls
2 ls is hashed (/sbin/ls)
3 $ type cd
4 cd is a shell builtin
```

This shows that the `ls` command is an executable, and the `cd` command is a shell built-in.

We can also use the `-a` flag to show all the ways the command can be interpreted.

```
1 $ type -a pwd
2 pwd is a shell builtin
3 pwd is /sbin/pwd
4 pwd is /bin/pwd
5 pwd is /usr/bin/pwd
6 pwd is /usr/sbin/pwd
```

Here we can see that the `pwd` command is a shell built-in, and is also present in multiple locations in the system. But if we run the `pwd` command, the shell built-in will be executed.

type is also useful when you are not sure whether to use `man` or `help` for a command. Generally for a shell built-in, `help` is used, and for an executable the `info` and the `man` pages are used.

**Types of Commands:** A command can be an **alias**, a **shell built-in**, a **shell function**, **keyword**, or an **executable**.

The `type` command will show which type the command is.

- ▶ **alias:** A command that is an alias to another command defined by the user or the system.
- ▶ **builtin:** A shell built-in command is a command that is built into the shell itself. It is executed internally by the shell. This is usually faster than an external command.
- ▶ **file:** An executable file that is stored in the file system. It has to be stored somewhere in the `PATH` variable.
- ▶ **function:** A shell function is a set of commands that are executed when the function is called.
- ▶ **keyword:** A keyword is a reserved word that is part of the shell syntax. It is not a command, but a part of the shell syntax.

**Exercise 1.2.11** Find the path of the `true` command using `which`. Find a short description of the `true` command using `whatis`. Is the executable you found actually the one that is executed when you run `true`? Check using `type true`

**Question 1.2.11** How to create aliases? How to make them permanent? How to unset them?

**Answer 1.2.11** `alias name='command'` will create an alias.

`unalias name` will unset the alias.

To make them permanent, add the alias to the `~/.bashrc` file.

The `~/.bashrc` file is a script that is executed whenever a new terminal is opened.

**Question 1.2.12** How to run the normal version of a command if it is aliased?

**Answer 1.2.12** `\command` will run the normal version of `command` if it is aliased.

**Question 1.2.13** What is the difference between `which`, `whatis`, `whereis`, `locate`, and `type`?

**Answer 1.2.13** Each of the commands serve a different purpose:

- ▶ `which` will show the path of the command that will be executed.
- ▶ `whatis` will show a short description of the command.
- ▶ `whereis` will show the location of the command, source files, and man pages.
- ▶ `locate` is used to find files by name.
- ▶ `type` will show how the command will be interpreted by the shell.

## 1.2.7 User Management

**whoami:**

The `whoami` command is used to print the username of the current user.

```
1 | $ whoami
2 | sayan
```

### **groups:**

The `groups` command is used to display the groups that the current user belongs to.

```
1 | $ groups
2 | sys wheel rfkill autologin sayan
```

### **passwd**

The `passwd` command is used to change the password of the current user. The root user can also change the password of other users.<sup>25</sup>

### **who:**

The `who` command is used to display the users who are currently logged in.

```
1 | $ who
2 | sayan    tty2          2024-05-22 13:49
3 | sayan    pts/0          2024-05-22 15:58 (:0)
4 | sayan    pts/1          2024-05-22 15:58 (tmux
   |           (1082933).%2)
5 | sayan    pts/2          2024-05-22 15:58 (tmux
   |           (1082933).%1)
6 | sayan    pts/3          2024-05-22 15:58 (tmux
   |           (1082933).%3)
7 | sayan    pts/4          2024-05-22 15:58 (tmux
   |           (1082933).%4)
8 | sayan    pts/5          2024-05-22 15:58 (tmux
   |           (1082933).%5)
9 | sayan    pts/6          2024-05-22 15:58 (tmux
   |           (1082933).%6)
10 | sayan   pts/7          2024-05-22 15:58 (tmux
   |           (1082933).%7)
11 | sayan   pts/8          2024-05-22 15:58 (tmux
   |           (1082933).%8)
```

25: This executable is a special one, as it is a setuid program. This will be discussed in detail in Section 1.4.6.

```

12 sayan    pts/9          2024-05-22 15:58 (tmux
   (1082933).%9)
13 sayan    pts/10         2024-05-22 17:58 (:0)
14 sayan    pts/11         2024-05-22 18:24 (tmux
   (1082933).%10)
15 sayan    pts/12         2024-05-22 18:24 (tmux
   (1082933).%11)

```

**Exercise 1.2.12** Run the `who` command on the system commands VM. What is the output?

**w:**

The `w` command is used to display the users who are currently logged in and what they are doing.

```

1 $ w
2 19:47:07 up 5:57, 1 user,  load average:
   0.77, 0.80, 0.68
3 USER     TTY      LOGIN@  IDLE   JCPU   PCPU
   WHAT
4 sayan    tty2     13:49   5:57m 19:10  21.82
   s dwm

```

This is different from the `who` command as it only considers the login shell. Here `dwm` is the window manager running on the `tty2` terminal.

## 1.2.8 Date and Time

**date:**

The `date` command is used to print formatted date and time information. Without any arguments, it prints the current date and time.

```

1 $ date
2 Mon May 20 06:23:07 PM IST 2024

```

We can specify the date and time to be printed using the `-d` flag.

```

1 | $ date -d "2020-05-20 00:30:45"
2 | Wed May 20 12:30:45 AM IST 2020
3 | $ date -d "2019-02-29"
4 | date: invalid date '2019-02-29'
5 | $ date -d "2020-02-29"
6 | Sat Feb 29 12:00:00 AM IST 2020

```

**Exercise 1.2.13** Why did we get an error when trying to print the date 2019-02-29?

We can also modify the format of the date and time using the `+` flag and different **format specifiers**. Some of the important format specifiers are listed in Table Table 1.6. Rest of the format specifiers can be found in the `date` manual page.

```

1 | $ date +"%Y-%m-%d %H:%M:%S"
2 | 2024-05-20 18:23:07
3 | $ date +"%A, %B %d, %Y"
4 | Monday, May 20, 2024

```

**Table 1.6:** Date Format Specifiers

Specifier	Description
\%Y	Year
\%m	Month
\%d	Day
\%H	Hour
\%M	Minute
\%S	Second
\%A	Full weekday name
\%B	Full month name
\%a	Abbreviated weekday name
\%b	Abbreviated month name

We can even mention relative dates and times using the `date` command.

```

1 $ date -d "next year"
2 Tue May 19 06:23:07 PM IST 2025
3 $ date -d "next month"
4 Thu Jun 20 06:23:07 PM IST 2024
5 $ date -d "tomorrow"
6 Tue May 21 06:23:07 PM IST 2024
7 $ date -d "yesterday"
8 Sun May 19 06:23:07 PM IST 2024

```

**cal:**

The `cal` command is used to print a calendar. By default, it prints the calendar of the current month.

```

1 $ cal
2      May 2024
3 Su Mo Tu We Th Fr Sa
4           1  2  3  4
5   5  6  7  8  9 10 11
6 12 13 14 15 16 17 18
7 19 20 21 22 23 24 25
8 26 27 28 29 30 31

```

We can specify the month and year to print the calendar of that month and year.

```

1 $ cal 2 2024
2      February 2024
3 Su Mo Tu We Th Fr Sa
4           1  2  3
5   4  5  6  7  8  9 10
6 11 12 13 14 15 16 17
7 18 19 20 21 22 23 24
8 25 26 27 28 29

```

There are multiple flags that can be passed to the `cal` command to display different types of calendars and of multiple months or of entire year.

**Remark 1.2.4** In Linux, there are sometimes multiple implementations of the same command.

For example, there are two implementations of the `cal` command, one in the `bsdmaintools` package, which is the **BSD** implementation and also includes another binary named `ncal` for printing the calendar in vertical format. The other implementation is in the `util-linux` package, which does not contain a `ncal` binary. The flags and the output of the `cal` command can differ between the two implementations.

**Question 1.2.14** How to print the current date and time in some custom format?

**Answer 1.2.14** `date -d today +\%Y-\%m-\%d` will print the current date in the format `YYYY-MM-DD`. The format can be changed by changing the format specifiers. The format specifiers are given in the `man date` page. The `-d today` can be dropped, but is mentioned to show that the date can be changed to any date. It can be strings like '`2024-01-01`' or '`5 days ago`' or '`yesterday`', etc.

---

These are some of the basic commands that are used in the terminal. Each of these commands has many more options and flags that can be used to customize their behavior. It is left as an exercise to the reader to explore the manual pages of these commands and try out the different options and flags.

Many of the commands that we have discussed here are also explained in the form of short videos on [Robert Elder's Youtube Channel](#).

# 1.3 Navigating the File System

## 1.3.1 What is a File System?

Unlike Windows which has different drive letters for different partitions, Linux follows a unified file structure. The filesystem hierarchy is a tree of directories and files<sup>26</sup>. The root<sup>27</sup> of the filesystem tree is the directory `/`. The basic filesystem hierarchy structure can be seen in Figure 1.6 and Table Table 1.7.

26: A non-directory is a leaf node of a tree.

27: The root of a tree is the first node from which the tree originates. A tree can have only one root.

But what does so many directories mean? What do they do? What is the purpose of each directory?

**Table 1.7:** Linux Filesystem Hierarchy

Directory Path	Description
<code>/</code>	Root directory
<code>/bin</code>	Essential command binaries
<code>/boot</code>	Static files of the bootloader
<code>/dev</code>	Device files
<code>/etc</code>	Host-specific system configuration
<code>/home</code>	User home directories
<code>/lib</code>	Essential shared libraries and kernel modules
<code>/media</code>	Mount point for removable media
<code>/mnt</code>	Mount point for mounting a filesystem temporarily
<code>/opt</code>	Add-on application software packages
<code>/proc</code>	Virtual filesystem providing process information
<code>/root</code>	Home directory for the root user
<code>/run</code>	Run-time variable data
<code>/sbin</code>	Essential system binaries
<code>/srv</code>	Data for services provided by the system
<code>/sys</code>	Kernel and system information
<code>/tmp</code>	Temporary files
<code>/usr</code>	Secondary hierarchy
<code>/var</code>	Variable data

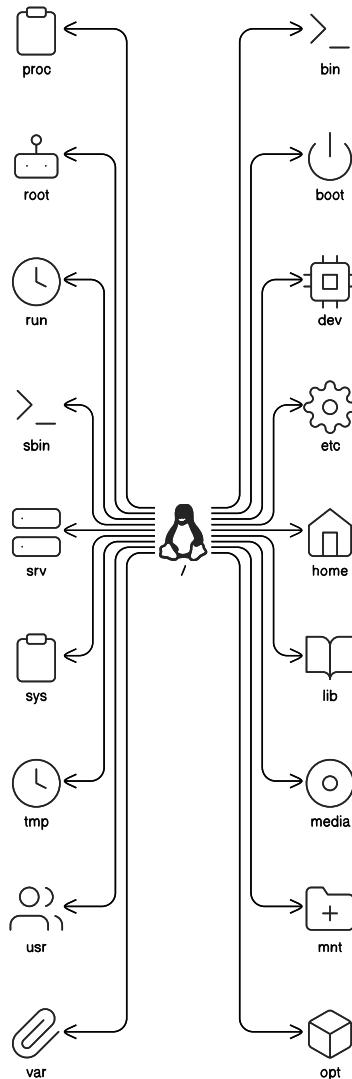


Figure 1.6: Linux Filesystem Hierarchy

Some directories do not store data on the disk, but are used to store information about the system. These directories are called *virtual* directories.

For example, the **/proc** directory is a virtual directory that provides information about the running processes. The **/sys** directory is another virtual directory that provides information about the system. The **/tmp** is a *volatile* directory whose data is deleted as soon as the system is turned off. The **/run** directory is another volatile directory that stores runtime data.

Rest directories are stored on the disk. The reason for having so many directories is to categorize the type of files they store. For example, all the executable binaries of different applications and utilities installed in the system is stored in **/bin** and **/sbin** directories. All the shared libraries installed on the system are stored in **/lib** directory. Sometimes some applications are installed in **/opt** directory which are not installed directly by the package manager.<sup>28</sup>

We also need to store the user's documents and files. This is done in the **/home** directory. Each user has their own directory in the **/home** directory. The root user's directory is **/root**. All the application's configuration files are stored in the user's home directory in the **/home** directory itself. This separation of application binary and per-user application settings helps people to easily change systems but keep their own **/home** directory constant and in turn, also all their application settings.

Some settings however needs to be applied system-wide and for all users. These settings are stored in the **/etc** directory. This directory contains all the system-wide configuration files.

To boot up the system, the bootloader needs some files. These files are stored in the **/boot** directory.<sup>29</sup> The bootloader is the first program that runs when

28: In Linux, you do not install applications by downloading them from the internet and running an installer like in Windows. You use a package manager to install applications. The package manager downloads the application from the internet and installs it on your system automatically. This way the package manager can also keep track of the installed applications and their dependencies and whether they should be updated. This is similar to the *Play Store* on mobile phones.

29: Modern systems use UEFI instead of BIOS to boot up the system. The bootloader is stored in the **/boot/EFI** directory or in the **/efi** directory directly.

the computer is turned on. It loads the operating system into memory and starts it.

Although the file system is a unified tree hierarchy, this doesn't mean that we cannot have multiple partitions on Linux: au contraire, it is easier to manage partitions on Unix. We simply need to mention which empty directory in the hierarchy should be used to mount a partition. As soon as that partition is mounted, it gets populated with the data stored on that disk with all the files and subdirectories, and when the device is unmounted the directory again becomes empty. Although a partition can be mounted on any directory, there are some dedicated folders in `/` as well for this purpose. For example, the `/mnt` directory is used to mount a filesystem temporarily, and the `/media` directory is used to mount removable media like USB drives, however it is not required to strictly follow this.

Finally, the biggest revelation in Linux is that, everything is a file. Not only are all the system configurations stored as **plain text** files which can be read by humans, but the processes running on your system are also stored as files in `proc`. Your kernel's interfaces to the applications or users are also simple files stored in `sys`. Biggest of all, even your hardware devices are stored as files in `dev`.<sup>30</sup>

30: Device files are not stored as normal files on the disk, but are special files that the kernel uses to communicate with the hardware devices. These are either **block** or **character** devices. They are used to read and write data to and from the hardware devices.

The `/usr` directory is a secondary hierarchy that contains subdirectories similar to those present in `/`. This was created as the olden system had started running out of disk space for the `/bin` and `/lib` directories. Thus another directory named `usr` was made, and subdirectiores like `/usr/bin` and `/usr/lib` were made to store half of the binaries and libraries. There wasn't however any rigid definition of which binary should go where. Modern day systems have

more than enough disk space to store everything on one partition, thus the **/bin** and **/lib** dont really exist any more. If they do, they are simply shortcuts<sup>31</sup> to the **/usr/bin** and **/usr/lib** directories which are still kept for *backwards compatibility*.

These can also be loosely classified into *sharable* and *non-sharable* directories and *static* and *variable* directories as shown in Table Table 1.8.

	Sharable	Non-sharable
Static	/usr, /opt	/etc, /boot
Variable	/var/spool	/tmp, /var/log

31: Shortcuts in Linux are called *symbolic links* or *symlinks*.

**Table 1.8:** Linux Filesystem Directory Classification

### 1.3.2 In Memory File System

Some file systems like **proc**, **sys**, **dev**, **run**, and **tmp** are not stored on the disk, but are stored in memory.

They have a special purpose and are used to store information about the system. These are called *virtual* directories.

These cannot be stored in a disk as it would be too slow to access them. Many of these files are very short lived yet are accessed very frequently. So these are stored in memory to speed up the access.

**/dev** and **/run** are mounted as **tmpfs** filesystems.

This can be seen by running the `mount` command or the `df` command.

```

1 $ mount
2 /dev/sdal on / type ext4 (rw,noatime)
3 devtmpfs on /dev type devtmpfs (rw,nosuid,size
   =4096k,nr_inodes=990693,mode=755,inode64)
```

```
4 tmpfs on /dev/shm type tmpfs (rw,nosuid,nodev,
      inode64)
5 devpts on /dev/pts type devpts (rw,nosuid,
      noexec,relatime,gid=5,mode=620,ptmxmode
      =000)
6 sysfs on /sys type sysfs (rw,nosuid,nodev,
      noexec,relatime)
7 securityfs on /sys/kernel/security type
      securityfs (rw,nosuid,nodev,noexec,
      relatime)
8 cgroup2 on /sys/fs/cgroup type cgroup2 (rw,
      nosuid,nodev,noexec,relatime,nsdelegate,
      memory_recursiveprot)
9 pstore on /sys/fs/pstore type pstore (rw,
      nosuid,nodev,noexec,relatime)
10 efivarfs on /sys/firmware/efi/efivars type
      efivarfs (rw,nosuid,nodev,noexec,relatime)
11 bpf on /sys/fs/bpf type bpf (rw,nosuid,nodev,
      noexec,relatime,mode=700)
12 configfs on /sys/kernel/config type configfs (
      rw,nosuid,nodev,noexec,relatime)
13 proc on /proc type proc (rw,nosuid,nodev,
      noexec,relatime)
14 tmpfs on /run type tmpfs (rw,nosuid,nodev,size
      =1590108k,nr_inodes=819200,mode=755,
      inode64)
15 systemd-1 on /proc/sys/fs/binfmt_misc type
      autofs (rw,relatime,fd=36,pgrp=1,timeout
      =0,minproto=5,maxproto=5,direct,pipe_ino
      =5327)
16 hugetlbfs on /dev/hugepages type hugetlbfs (rw
      ,nosuid,nodev,relatime,pagesize=2M)
17 mqueue on /dev/mqueue type mqueue (rw,nosuid,
      nodev,noexec,relatime)
18 debugfs on /sys/kernel/debug type debugfs (rw,
      nosuid,nodev,noexec,relatime)
19 tracefs on /sys/kernel/tracing type tracefs (
      rw,nosuid,nodev,noexec,relatime)
20 fusectl on /sys/fs/fuse/connections type
      fusectl (rw,nosuid,nodev,noexec,relatime)
21 systemd-1 on /data type autofs (rw,relatime,fd
```

```

1      =47,pgrp=1,timeout=60,minproto=5,maxproto
2      =5,direct,pipe_ino=2930)
3 tmpfs on /tmp type tmpfs (rw,noatime,inode64)
4 /dev/sda4 on /efi type vfat (rw,relatime,fmask
5     =0137,dmask=0027,codepage=437,iocharset=
6     ascii,shortname=mixed,utf8,errors=remount-
7     ro)
8 /dev/sda2 on /home type ext4 (rw,noatime)
9 binfmt_misc on /proc/sys/fs/binfmt_misc type
10    binfmt_misc (rw,nosuid,nodev,noexec,
11    relatime)
12 tmpfs on /run/user/1000 type tmpfs (rw,nosuid,
13     nodev,relatime,size=795052k,nr_inodes
14     =198763,mode=700,uid=1000,gid=1001,inode64
15     )
16 portal on /run/user/1000/doc type fuse.portal
17    (rw,nosuid,nodev,relatime,user_id=1000,
18    group_id=1001)
19 /dev/sdb3 on /data type ext4 (rw,noatime,x-
20     systemd.automount,x-systemd.idle-timeout=1
21     min)

```

Here we can see that the **/dev** directory is mounted as a **devtmpfs** filesystem. The **/run** directory is mounted as a **tmpfs** filesystem. The **/proc** directory is mounted as a **proc** filesystem. The **/sys** directory is mounted as a **sysfs** filesystem.

These are all virtual filesystems that are stored in memory.

### **proc:**

Proc is an old filesystem that is used to store information about the running processes. The **/proc** directory contains a directory for each running process. The directories are named as the process id of the process.

```

1 $ ls -l /proc | head
2 total 0
3 dr-xr-xr-x  9 root  root  0 May 23 13:01 1

```

```

4 dr-xr-xr-x  9 root  root  0 May 23 13:01 100
5 dr-xr-xr-x  9 sayan sayan 0 May 23 13:06 1004
6 dr-xr-xr-x  9 sayan sayan 0 May 23 13:06 1009
7 dr-xr-xr-x  9 root  root  0 May 23 13:01 102
8 dr-xr-xr-x  9 root  sayan 0 May 23 13:06 1029
9 dr-xr-xr-x  9 sayan sayan 0 May 23 13:06 1038
10 dr-xr-xr-x  9 sayan sayan 0 May 23 13:06 1039
11 dr-xr-xr-x  9 sayan sayan 0 May 23 13:06 1074

```

These folders are simply for information and do not store any data. This is why they have a size of 0. Each folder is owned by the user who started the process.

Inside each of these directories, there are files that contain information about the process.

You can enter the folder of a process that is started by you and see the information about the process.

```

1 $ cd /proc/301408
2 $ ls -l | head -n15
3 total 0
4 -r--r--r--  1 sayan sayan 0 May 23 16:55
   arch_status
5 dr-xr-xr-x  2 sayan sayan 0 May 23 16:55 attr
6 -rw-r--r--  1 sayan sayan 0 May 23 16:55
   autogroup
7 -r-----  1 sayan sayan 0 May 23 16:55 auxv
8 -r--r--r--  1 sayan sayan 0 May 23 16:55
   cgroup
9 --w-----  1 sayan sayan 0 May 23 16:55
   clear_refs
10 -r--r--r--  1 sayan sayan 0 May 23 16:55
    cmdline
11 -rw-r--r--  1 sayan sayan 0 May 23 16:55 comm
12 -rw-r--r--  1 sayan sayan 0 May 23 16:55
    coredump_filter
13 -r--r--r--  1 sayan sayan 0 May 23 16:55
    cpu_resctrl_groups
14 -r--r--r--  1 sayan sayan 0 May 23 16:55
    cpuset

```

```

15 lrwxrwxrwx 1 sayan sayan 0 May 23 16:55 cwd
    -> /home/sayan/docs/projects/sc-handbook
16 -r----- 1 sayan sayan 0 May 23 16:55
    environ
17 lrwxrwxrwx 1 sayan sayan 0 May 23 13:41 exe
    -> /usr/bin/entr

```

Here you can see that the command line of the process is stored in the **cmdline** file. Here the process is of a command called **entr**.

You can also see the **current working directory** (cwd) of the process.

There are some other files in the **/proc** directory that contain information about the system.

- ▶ **cpuinfo** - stores cpu information.
- ▶ **version** - stores system information, content similar to `uname -a` command.
- ▶ **meminfo** - Diagnostic information about memory. Check `free` command.
- ▶ **partitions** - Disk partition information. Check `df`.
- ▶ **kcore** - The astronomical size ( $2^{47}$  bits) tells the maximum virtual memory (47 bits) the current Linux OS is going to handle.

### sys:

Sys is a newer filesystem that is used to store information about the system. It is neatly organized and is easier to navigate than proc. This highly uses symlinks to organize the folders while maintaining redundancy as well.<sup>32</sup>

Try running the following code snippet in a terminal if you have a caps lock key on your keyboard and are running linux directly on your bare-metal.<sup>33</sup>

```

1 $ cd /sys/class/leds
2 $ echo 1 | sudo tee *capslock/brightness

```

32: A symlink is a special type of file that points to another file or directory. It is similar to a shortcut in Windows. This will be discussed in detail in Section 1.6.

33: This will only work on your own system, not on the system commands VM, since you do not have the privilege to modify the files there. Make sure you have the ability to run commands as root and are able to use `sudo`. It is also unlikely to work on a virtual machine. It will also not work on linux systems older than 2.6.

If you are running a linux system directly on your hardware, you will see the caps lock key light up. Most modern keyboards will quickly turn off the light again as the capslock is technically not turned on, only the led was turned on manually by you.

### **/sys vs /proc:**

34: Unix System V is one of the first commercial versions of the Unix operating system. It was originally developed by AT&T and first released in 1983.

35: BSD, or Berkeley Software Distribution, is a Unix-like operating system that was developed at the University of California, Berkeley. It was first released in 1977 and was based on the original Unix source code from AT&T. BSD is not linux, it is a totally different kernel, with similar core utils to GNU.

The /proc tree originated in System V Unix <sup>34</sup>, where it only gave information about each running process, using a /proc/\$PID/ format. Linux greatly extended that, adding all sorts of information about the running kernel's status. In addition to these read-only information files, Linux's /proc also has writable virtual files that can change the state of the running kernel. BSD <sup>35</sup> type OSes generally do not have /proc at all, so much of what you find under proc is non-portable.

The intended solution for this mess in Linux's /proc is /sys. Ideally, all the non-process information that got crammed into the /proc tree should have moved to /sys by now, but historical inertia has kept a lot of stuff in /proc. Often there are two ways to effect a change in the running kernel: the old /proc way, kept for backwards compatibility, and the new /sys way that you're supposed to be using now.

### **1.3.3 Paths**

Whenever we open a terminal on a Linux system, we are placed in a directory. This is called the *current working directory*. All shells and applications have a current working directory from where they are launched.

To refer to and identify the directory you are talking about, we use a **path**.

**Definition 1.3.1 (Path)** Path is a traversal in the filesystem tree. It is a way to refer to a file or directory.

### Absolute Path:

The traversal to the directory from the root directory is called the **absolute path**. For example, if we want to refer to the directory named **alex** inside the directory **home** in the root of the file system, then it is qualified as:

```
1 | /home/alex
```

### Relative Path:

The traversal to the directory from the current working directory is called the **relative path**. For example, if we want to refer to the directory named **alex** inside the directory **home** from the **/usr/share** directory, then it will be qualified as:

```
1 | ../../home/alex
```

**Remark 1.3.1** The **..** in the path refers to the parent directory. It is used in relative paths to refer to directories whose path requires travelling up the tree.

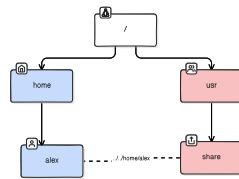


Figure 1.7: Relative Path

### 1.3.4 Basic Commands for Navigation

The file system can be navigated in the Linux command line using the following commands:

- ▶ **pwd**: Print the current working directory
- ▶ **ls**: List the contents of the current directory
- ▶ **cd**: Change the current working directory
- ▶ **mkdir**: Create a new directory

- ▶ **rmdir:** Remove a directory
- ▶ **touch:** Create a new file
- ▶ **rm:** Remove a file
- ▶ **pushd:** Push the current directory to a stack
- ▶ **popd:** Pop the current directory from a stack<sup>36</sup>

36: **pushd** and **popd** are useful for quickly switching between directories in scripts.

More details about these commands can be found in their respective man pages. For example, to find more about the **ls** command, you can type **man ls**.

**Question 1.3.1** What is the command to list the contents of the current directory?

**Answer 1.3.1** **ls**

**Question 1.3.2** What is the command to list the contents of the current directory including hidden files?

**Answer 1.3.2** **ls -a**

**Question 1.3.3** What is the command to list the contents of the current directory in a long list format? (show permissions, owner, group, size, and time)

**Answer 1.3.3** **ls -l**

**Question 1.3.4** What is the command to list the contents of the current directory in a long list format and also show hidden files?

**Answer 1.3.4** `ls -al` or `ls -la` or `ls -l -a` or  
`ls -a -l`

**Question 1.3.5** The output of `ls` gives multiple files and directories in a single line. How can you make it print one file or directory per line?

**Answer 1.3.5** `ls -1`

This can also be done by passing the output of `ls` to `cat` or storing the output of `ls` in a file and then using `cat` to print it. We will see these in later weeks.<sup>37</sup>

37: that is a one, not an L

## 1.4 File Permissions

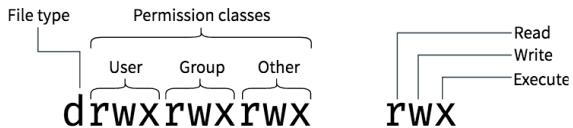


Figure 1.8: File Permissions

**Definition 1.4.1 (File Permissions)** File permissions define the access rights of a file or directory. There are three basic permissions: read, write, and execute. These permissions can be set for the owner of the file, the group of the file, and others.

We have already briefly seen how to see the permissions of a file using the `ls -l` command.

```

1 $ touch hello.txt
2 $ mkdir world
3 $ ls -l
4 total 4
5 -rw-r--r-- 1 sayan sayan    0 May 21 15:20
       hello.txt
6 drwxr-xr-x 2 sayan sayan 4096 May 21 15:21
       world

```

Here, the first column of the output of `ls -l` shows the permissions of the file or directory. As seen in Figure 1.8, the permissions are divided into four parts:

- ▶ The first character shows the type of the file.  
- for a regular file and `d` for a directory and more.<sup>38</sup>
- ▶ The next three characters show the permissions for the owner of the file.
- ▶ The next three characters show the permissions for the group of the file.

38: There are other types of files as well, like `l` for a symbolic link, `c` for a character device, `b` for a block device, `s` for a socket, and `p` for a pipe. These will be discussed later.

- The last three characters show the permissions for others.

**Definition 1.4.2 (Owner)** Although this can be changed, the owner of a file is usually the user who created it. All files in the filesystem have an owner. This is symbolically coded as **u**.

**Definition 1.4.3 (Group)** The group of a file is usually the group of the user who created it. But it can also be changed to any other existing group in the system. All users in the group <sup>a</sup> have the same permissions on the file. This is symbolically coded as **g**.

<sup>a</sup> except the owner of the file

**Definition 1.4.4 (Others)** Others are all the users who are not the owner of the file and are not in the group of the file. This is symbolically coded as **o**.

There are three actions that can be performed on a file: read, write, and execute.

- **Read:** The read permission allows the file to be read. This is symbolically coded as **r**.
- **Write:** The write permission allows the file to be modified. This is symbolically coded as **w**.
- **Execute:** The execute permission allows the file to be executed.<sup>39</sup> This is symbolically coded as **x**.

39: Executing a file means running the file as a program. For a directory, the execute permission allows the directory to be traversed into.

These however, have different meanings for files and directories.

### 1.4.1 Read

- ▶ For a file, the read permission allows the file to be read. You can use commands like `cat` or `less` to read the contents of the file if the user has **read** permissions.
- ▶ For a directory, the read permission allows the directory to be listed using `ls`.

### 1.4.2 Write

40: Redirection is a way to send the output of a command to a file.

- ▶ For a file, the write permission allows the file to be modified. You can use commands like `echo` along with redirection<sup>40</sup> or a text editor like `vim` or `nano` to write to the file if the user has **write** permissions.
- ▶ For a directory, the write permission allows the directory to be modified. You can create, delete, or rename files in the directory if the user has **write** permissions.

### 1.4.3 Execute

- ▶ For a file, the execute permission allows the file to be executed. This is usually only needed for special files like executables, scripts, or libraries. You can run the file as a program if the user has **execute** permissions.
- ▶ For a directory, the execute permission allows the directory to be traversed into. You can change to the directory if the user has **execute** permissions using `cd`. You can also only long-list the contents of the directory if the user has **execute** permissions on that directory.

### 1.4.4 Interesting Caveats

This causes some interesting edge-cases that one needs to be familiar with.

#### Cannot modify a file? Think again!

If you have **write** and **execute** permissions on a directory, even if you do not have **write** permission on a file inside the directory, you can **delete** the file due to your **write** permission on the directory, and then re-create the modified version of the file with the same name. But if you try to simply modify the file directly, you will get permission error.

```

1 $ mkdir test
2 $ cd test
3 $ echo "hello world" > file1
4 $ chmod 400 file1      # 400 means read
                         permission only
5 $ cat file1
6 hello world
7 $ echo "hello universe" > file1 # unable to
                         write
8 -bash: file1: Permission denied
9 $ rm file1 # can remove as we have write
             permission on folder
10 rm: remove write-protected regular file '
               file1'? y
11 $ echo "hello universe" > file1 # can create
               new file
12 $ cat file1
13 hello universe

```

However, this only works on files. You cannot remove a directory if you do not have **write** permission on the directory, even if you have **write** permission on its parent directory.

#### Can list names but not metadata?

If you have **read** permission on a directory but not **execute** permission, you cannot traverse into the directory, but you can still use `ls` to list the contents of the directory. However, you cannot use `ls -l` to long-list the contents of the directory. That is, you only have access to the name of the files inside, not their metadata.

```

1 $ mkdir test
2 $ touch test/1 test/2
3 $ chmod 600 test # removing execute
      permission from folder
4 $ ls test # we can still list the files due
      to read permission
5  2
6 $ ls -l test # but cannot long-list the
      files
7 ls: cannot access 'test/2': Permission
      denied
8 ls: cannot access 'test/1': Permission
      denied
9 total 0
10 -????????? ? ? ? ? ? ? 1
11 -????????? ? ? ? ? ? ? 2

```

### Cannot list names but can traverse?

If you have **execute** permission on a directory but not **read** permission, you can traverse into the directory but you cannot list the contents of the directory.

```

1 $ mkdir test
2 $ touch test/1 test/2
3 $ chmod 300 test # removing read permission
      from folder
4 $ ls test # we cannot list the files
5 ls: cannot open directory 'test': Permission
      denied
6 $ cd test # but we can traverse into the
      folder
7 $ pwd

```

```
8 | /home/sayan/test
```

### Subdirectories with all permissions, still cannot access?

If you have all the permissions to a directory, but dont have **execute** permission on its parent directory, you cannot access the subdirectory, or even list its contents.

```
1 | $ mkdir test
2 | $ mkdir test/test2 # subdirectory
3 | $ touch test/test2/1 # file inside
   subdirectory
4 | $ chmod 700 test/test2 # all permissions to
   subdirectory
5 | $ chmod 600 test # removing execute
   permission from parent directory
6 | $ ls test
7 | test2
8 | $ cd test/test2 # cannot access subdirectory
9 | -bash: cd: test/test2: Permission denied
10 | $ ls test/test2 # cannot even list contents
   of subdirectory
11 | ls: cannot access 'test/test2': Permission
   denied
```

## 1.4.5 Changing Permissions

The permissions of a file can be changed using the **chmod** command.

### Synopsis:

```
1 | chmod [OPTION]... MODE[,MODE]... FILE...
2 | chmod [OPTION]... OCTAL-MODE FILE...
```

**OCTAL-MODE** is a 3 or 4 digit octal number where the first digit is for the owner, the second digit is for the group, and the third digit is for others. We will

discuss how the octal representation of permissions is calculated in the next section.

The **MODE** can be in the form of **ugoa+-=rwxXst** where:

- ▶ **u** is the user who owns the file
- ▶ **g** is the group of the file
- ▶ **o** is others
- ▶ **a** is all
- ▶ **+** adds the permission
- ▶ **-** removes the permission
- ▶ **=** sets the permission
- ▶ **r** is read
- ▶ **w** is write
- ▶ **x** is execute
- ▶ **X** is execute only if its a directory or already has execute permission.
- ▶ **s** is setuid/setgid
- ▶ **t** is restricted deletion flag or sticky bit

We are already familiar with what **r**, **w**, and **x** permissions mean, but what are the other permissions?

#### 1.4.6 Special Permissions

**Definition 1.4.5 (SetUID/SetGID)** The setuid and setgid bits are special permissions that can be set on executable files. When an executable file has the setuid bit set, the file will be executed with the privileges of the owner of the file. When an executable file has the setgid bit set, the file will be executed with the privileges of the group of the files.

**SetUID:**

This is useful for programs that need to access system resources that are only available to the owner or group of the file.

A very notable example is the `passwd` command. This command is used to set the password of an user. Although changing password of a user is a priviledged action that only the root user can do, the `passwd` command can be run by any user to change *their* password. This is possible due to the setuid bit set on the `passwd` command. When the `passwd` command is run, it is run with the privileges of the root user, and thus can change the password of that user.

You can check this out by running `ls -l /usr/bin/passwd` and seeing the s in the permissions.

```
1 $ ls -l /usr/bin/passwd
2 -rwsr-xr-x 1 root root 80912 Apr  1 15:49 /
   usr/bin/passwd
```

### SetGID:

The behaviour of **SetGID** is similar to **SetUID**, but the file is executed with the privileges of the group of the file.

However, **SetGID** can also be applied to a directory. When a directory has the **SetGID** bit set, all the files and directories created inside that directory will inherit the group of the directory, not the group of the user who created the file or directory. This is highly useful when you have a directory where multiple users need to work on the same files and directories, but you want to restrict the access to only a certain group of users. The primary group of each user is different from each other, but since they are also part of another group (which is the group owner of the directory) they are able to read and write the files present in the directory. However, if

the user creates a file in the directory, the file will be owned by the user's primary group, not the group of the directory. So other users would not be able to access the file. This is fixed by the **SetGID** bit on the directory.

```
1 $ mkdir test
2 $ ls -ld test # initially the folder is
   owned by the user's primary group
3 drwxr-xr-x 2 sayan sayan 4096 May 22 16:27
   test
4 $ chgrp wheel test # we change the group of
   the folder to wheel, which is a group that
   the user is part of
5 $ ls -ld test
6 drwxr-xr-x 2 sayan wheel 4096 May 22 16:27
   test
7 $ whoami # this is the current user
8 sayan
9 $ groups # this is the users groups, first
   one is its primary group
10 sayan wheel
11 $ touch test/file1 # before setting the
   SetGID bit, a new file will have group
   owner as the primary group of the user
   creating it
12 $ ls -l test/file1 # notice the group owner
   is sayan
13 -rw-r--r-- 1 sayan sayan 0 May 22 16:29 test
   /file1
14 $ chmod g+s test # we set the SetGID bit on
   the directory
15 $ ls -ld test # now the folder has a s in
   the group permissions
16 drwxr-sr-x 2 sayan wheel 4096 May 22 16:29
   test
17 $ touch test/file2 # now if we create
   another new file, it will have the group
   owner as the group of the directory
18 $ ls -l test/file2 # notice the group owner
   is wheel
```

```
19 -rw-r--r-- 1 sayan wheel 0 May 22 16:29 test
    /file2
```

### Restricted Deletion Flag or Sticky Bit:

The restricted deletion flag or sticky bit is a special permission that can be set on directories.

Historically, this bit was to be applied on executable files to keep the program in memory after it has finished executing. This was done to speed up the execution of the program as the program would not have to be loaded into memory again. This was called **sticky bit** because the program would stick in memory.<sup>41</sup>

However, this is no longer how this bit is used.

When the sticky bit is set on a directory, only the owner of the file, the owner of the directory, or the root user can delete or rename files in the directory.

This is useful when you have a directory where multiple users need to write files, but you want to restrict the deletion of files to only the owner of the file or the owner of the directory.

The most common example of this is the `/tmp` directory. The `/tmp` directory is a directory where temporary files are stored. You want to let any user create files in the `/tmp` directory, but you do not want any user to delete files created by other users.

<sup>41</sup>: The part of memory where the program's text segment is stored is called the *swap*.

```
1 $ ls -ld /tmp
2 drwxrwxrwt 20 root root 600 May 22 16:43 /
    tmp
```

**Exercise 1.4.1** Log into the system commands VM and cd into the `/tmp` directory. Create a file

Octal:	0	6	4	0
Binary:	000	110	100	000
Symbolic:	S t	r w X	r w X	r w X
Special attributes	User (u)	Group (g)	Other (o)	All (a)
				All (a)

Figure 1.9: Octal Permissions

in the `/tmp` directory. Try to find if there are files created by other users in the `/tmp` directory using `ls -l` command. If there are files created by other users, try to delete them.<sup>a</sup>

<sup>a</sup> You can create a file normally, or using the `mktemp` command.

### 1.4.7 Octal Representation of Permissions

42: If the octal is 4 digits, the first digit is for special permissions like setuid, setgid, and sticky bit.

The permissions of a file for the file's owner, group, and others can be represented as a 3 or 4 digit octal number.<sup>42</sup> Each of the octal digits is the sum of the permissions for the owner, group, and others.

- ▶ Read permission is represented by 4
- ▶ Write permission is represented by 2
- ▶ Execute permission is represented by 1

Thus if a file has read, write, and execute permissions for the owner, read and execute permissions for the group, and only read permission for others, the octal representation of the permissions would be 754.

**Table 1.9:** Octal Representation of Permissions

Octal	Read	Write	Execute	Representation	Description
0	0	0	0	—	No permissions
1	0	0	1	-x	Execute only
2	0	1	0	-w-	Write only
3	0	1	1	-wx	Write and execute
4	1	0	0	r-	Read only
5	1	0	1	r-x	Read and execute
6	1	1	0	rw-	Read and write
7	1	1	1	rwx	Read, write, and execute

The octal format is usually used more than the symbolic format as it is easier to understand and remember and it is more concise.

```

1 $ chmod 754 myscript.sh # this sets the
    permissions of myscript.sh to rwxr-xr--
2 $ ./myscript.sh
3 Hello World!

```

However, if you want to add or remove a permission without changing the other permissions, the symbolic format is more useful.

```

1 $ chmod u+x myscript.sh # this adds execute
    permission to the owner of myscript.sh
2 $ ./myscript.sh
3 Hello World!

```

**Question 1.4.1** How to list the permissions of a file?

**Answer 1.4.1** `ls -l`

The permissions are the first 10 characters of the output.

`stat -c \%A filename` will list only the permis-

sions of a file.

There are other format specifiers of `stat` to show different statistics which can be found in `man stat`.

**Question 1.4.2** How to change permissions of a file? Let's say we want to change `file1`'s permissions to `rwxr-xr--`. What is the octal form of that?

**Answer 1.4.2** `chmod u=rwx,g=rx,o=r file1` will change the permissions of `file1`.

The octal form of `rwxr-xr--` is 754.

So we can also use `chmod 754 file1`

Providing the octal is same as using = to set the permissions.

We can also use + to add permissions and - to remove permissions.

## 1.5 Types of Files

We had briefly seen that the output of `ls -l` shows the type of the file as the first character of the permissions.

There are 7 types of files in a linux file system as shown in Table Table 1.10.

**Table 1.10:** Types of Files

Type	Symbol
Regular Files	-
Directories	d
Symbolic Links	l
Character Devices	c
Block Devices	b
Named Pipes	p
Sockets	s

### 1.5.1 Regular Files

Regular files are the most common type of file. Almost all files are regular files. Scripts and executable binaries are also regular files. All the configuration files of the system are regular files as well. The regular files are actually the only files that contain data and are stored on the disk.

### 1.5.2 Directories

Directories are files that contain a list of other files. Directories do not contain data, they contain references to other files. Usually the size of a directory is equal to the block size of the filesystem. Directories have some special permissions that are different from regular files as discussed in Section 1.4.

### 1.5.3 Symbolic Links

Symbolic links are files that point to other files. They only consume the space of the path they are pointing to. Symlinks<sup>43</sup> are useful to create shortcuts to files or directories. They are dependent on the original file and will stop working if the original file is deleted or moved. They are discussed in detail in Section 1.6.

43: Symlinks is short for symbolic links.

There are another type of links called **hard links**. However, hard links are not files, they are pointers to the same inode. They do not consume extra space, and are not dependent on the original file. Hard links do not have a separate type, they are just regular files.

### 1.5.4 Character Devices

Character devices are files that represent devices that are accessed as a stream of bytes. For example the keyboard, mouse, webcams, and most USB devices are character devices. These are not real files stored on the disk, but are files that represent devices. They can interacted with like a file using the read and write system calls to interact with the hardware directly. These files are made available by the kernel and are stored in the /dev directory. Any read/write operation on a character device is monitored by the kernel and the data is sent to the device.

```
1 $ cd /dev/input
2 $ ls -l
3 total 0
4 drwxr-xr-x 2 root root      220 May 22 13:49 by
   -id
5 drwxr-xr-x 2 root root      420 May 22 13:49 by
   -path
6 crw-rw---- 1 root input 13, 64 May 22 13:49
   event0
7 crw-rw---- 1 root input 13, 65 May 22 13:49
   event1
8 crw-rw---- 1 root input 13, 74 May 22 13:49
   event10
9 crw-rw---- 1 root input 13, 75 May 22 13:49
   event11
10 crw-rw---- 1 root input 13, 76 May 22 13:49
    event12
11 crw-rw---- 1 root input 13, 77 May 22 13:49
    event13
12 crw-rw---- 1 root input 13, 78 May 22 13:49
    event14
13 crw-rw---- 1 root input 13, 79 May 22 13:49
    event15
14 crw-rw---- 1 root input 13, 80 May 22 13:49
    event16
```

```
15 crw-rw---- 1 root input 13, 81 May 22 13:49
    event17
16 crw-rw---- 1 root input 13, 82 May 22 13:49
    event18
17 crw-rw---- 1 root input 13, 83 May 22 13:49
    event19
18 crw-rw---- 1 root input 13, 66 May 22 13:49
    event2
19 crw-rw---- 1 root input 13, 84 May 22 13:49
    event20
20 crw-rw---- 1 root input 13, 67 May 22 13:49
    event3
21 crw-rw---- 1 root input 13, 68 May 22 13:49
    event4
22 crw-rw---- 1 root input 13, 69 May 22 13:49
    event5
23 crw-rw---- 1 root input 13, 70 May 22 13:49
    event6
24 crw-rw---- 1 root input 13, 71 May 22 13:49
    event7
25 crw-rw---- 1 root input 13, 72 May 22 13:49
    event8
26 crw-rw---- 1 root input 13, 73 May 22 13:49
    event9
27 crw-rw---- 1 root input 13, 63 May 22 13:49
    mice
28 crw-rw---- 1 root input 13, 32 May 22 13:49
    mouse0
29 crw-rw---- 1 root input 13, 33 May 22 13:49
    mouse1
```

Here the `event` and `mouse` files are character devices that represent input devices like the keyboard and mouse. Note the `c` in the permissions, which indicates that these are character devices.

### 1.5.5 Block Devices

Block devices are files that represent devices that are accessed as a block of data. For example hard

drives, SSDs, and USB drives are block devices. These files also do not store actual data on the disk, but represent devices. Any block file can be mounted as a filesystem. We can interact with block devices using the `read` and `write` system calls to interact with the hardware directly. For example, the `/dev/sda` file represents the first hard drive in the system.

44: The `dd` command is a powerful tool that can be used to copy and convert files. It is acronym of *data duplicator*. However, it is also known as the *disk destroyer* command, as it can be used to overwrite the entire disk if you are not careful with which disk you are writing the image to.

45: ISO file

This makes it easy to write an image to a disk directly using the `dd` command.<sup>44</sup>

The following example shows how we can use the `dd` command to write an image<sup>45</sup> to a USB drive. It is this easy to create a bootable USB drive for linux.

```
$ dd if=~/Downloads/archlinux.iso of=/dev/sdb bs=4M status=progress
```

Here `if` is the input file, `of` is the output file, `bs` is the block size, and `status` is to show the progress of the operation.

**Warning 1.5.1** Be very careful when using the `dd` command. Make sure you are writing to the correct disk. Writing to the wrong disk can cause data loss.

## 1.5.6 Named Pipes

46: Also known as FIFOs

47: and vice versa

Named pipes<sup>46</sup> are files that are used for inter-process communication. They do not store the data that you write to them, but instead pass the data to another process. A process can only write data to a named pipe if another process is reading from the named pipe.<sup>47</sup>

```
1 $ mkfifo pipe1
2 $ ls -l pipe1
3 prw-r--r-- 1 sayan sayan 0 May 22 18:22 pipe1
```

Here the p in the permissions indicates that this is a named pipe. If you now try to write to the named pipe, the command will hang until another process reads from the named pipe. Try the following in two different terminals:

**Terminal 1:**

```
1 $ echo "hello" > pipe1
```

**Terminal 2:**

```
1 $ cat pipe1
```

You will notice that whichever command you run first will hang until the other command is run.

### 1.5.7 Sockets

Sockets are a special file type, similar to TCP/IP sockets, providing inter-process networking protected by the file system's access control.

This is similar to named pipes, but the difference is that named pipes are meant for IPC between processes in the same machine, whereas sockets can be used for communication across machines.

Try out the following in two different terminals:

**Terminal 1:**

```
1 $ nc -lU socket.sock
```

**Terminal 2:**

```
1 $ echo "hello" | nc -U socket.sock
```

Notice here, that if you run the command in terminal 2 first, it will error out with the text:

```
1 | nc: socket.sock: Connection refused
```

Only if we run them in correct order can you see the message "hello" being printed in terminal 1.<sup>48</sup>

You can press **Ctrl+C** to stop the nc command in both terminals.

<sup>48</sup>: The nc command is the netcat command. It is a powerful tool for network debugging and exploration. It can be used to create sockets, listen on ports, and send and receive data over the network. This will be discussed in more detail in the networking section and in the **Modern Application Development** course.

The bytes at the start of a file used to identify the type of file are called the **magic bytes**.

More details can be found at: [https://en.wikipedia.org/wiki/List\\_of\\_file\\_signatures](https://en.wikipedia.org/wiki/List_of_file_signatures)

```
1 | $ file /etc/passwd
2 | /etc/passwd: ASCII text
3 | $ file /bin/bash
4 | /bin/bash: ELF 64-bit LSB pie executable, x86
   -64, version 1 (SYSV), dynamically linked,
   interpreter /lib64/ld-linux-x86-64.so.2,
   BuildID[sha1]=165
   d3a5ffe12a4f1a9b71c84f48d94d5e714d3db, for
   GNU/Linux 4.4.0, stripped
```

**Question 1.5.1** What types of files are possible in a linux file system?

**Answer 1.5.1** There are 7 types of files in a linux file system:

- ▶ Regular Files (starts with -)
- ▶ Directories (starts with d)
- ▶ Symbolic Links (starts with l)
- ▶ Character Devices (starts with c)
- ▶ Block Devices (starts with b)
- ▶ Named Pipes (starts with p)
- ▶ Sockets (starts with s)

**Question 1.5.2** How to know what kind of file a file is? Can we determine using its extension? Can we determine using its contents? What does *MIME* mean? How to get that?

**Answer 1.5.2** The `file` command can be used to determine the type of a file.

The extension of a file does not determine its type.

The contents of a file can be used to determine its type.

MIME stands for Multipurpose Internet Mail Extensions.

It is a standard that indicates the nature and format of a document.

`file -i filename` will give the MIME type of `filename`.

## 1.6 Inodes and Links

### 1.6.1 Inodes

**Definition 1.6.1 (Inodes)** An inode is an index node. It serves as a unique identifier for a specific piece of metadata on a given filesystem.

Whenever you run `ls -l` and see all the details of a file, you are seeing the metadata of the file. These metadata, however, are not stored in the file itself. These data about the files are stored in a special data structure called an **inode**.

Each inode is stored in a common table and each filesystem mounted to your computer has its own inodes. An inode number may be used more than once but never by the same filesystem. The filesystem id combines with the inode number to create a unique identification label.

You can check how many inodes are used in a filesystem using the `df -i` command.

```
1 $ df -i
2 Filesystem      Inodes   IUsed   IFree  IUse%
3           Mounted on
4 /dev/sda1        6397952  909213  5488739  15%
5           /
6 /dev/sda4          0       0       0       -
7           /efi
8 /dev/sda2        21569536 2129841 19439695  10%
9           /home
10 /dev/sdb3         32776192   2380  32773812   1%
11           /data
12 $ df
13 Filesystem  1K-blocks  Used Available
14           Use% Mounted on
15 /dev/sda1    100063312 63760072 31174076
16           68% /
```

```

10 | /dev/sda4      1021952    235760    786192
   |     24% /efi
11 | /dev/sda2      338553420  273477568  47805068
   |     86% /home
12 | /dev/sdb3      514944248  444194244  44518760
   |     91% /data

```

You can notice the number of inodes present, number of inodes used, and number of inodes that are free. The **IUse%** column shows the percentage of inodes used. This however, does not mean how much of space is used, but how many files can be created.

Observe that although the `/data` partition has only 1% of inodes used, it has 91% of space used. This is because the files in the `/data` partition are large files, and thus the number of inodes used is less. Remember that a file will take up one inode, no matter how large it is. But the space it takes up will be the size of the file.

We can also see the inode number of a file using the `ls -i` command.

```

1 | $ ls -i
2 | 1234567 file1
3 | 1234568 file2
4 | 1234569 file3

```

Here the first column is the **inode** number of the file.

**Remark 1.6.1** The inode number is unique only within the filesystem. If you copy a file from one filesystem to another, the inode number will change.

## 1.6.2 Separation of Data, Metadata, and Filename

In UNIX systems, the data of a file and the metadata of a file are stored separately. The inodes are stored in a inode-array or table, and contain the metadata of the file and the pointer to it in the storage block. These metadata can be retrieved using the `stat` system call.<sup>49</sup>

- 49: A system call is a request in a operating system made via a software interrupt by an active process for a service performed by the kernel. The diagram in Figure 1.10 shows how system calls work.

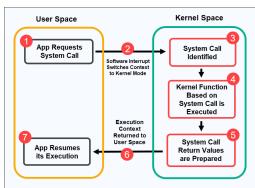


Figure 1.10: System Calls

```

1 $ stat /etc/profile
2   File: /etc/profile
3   Size: 993          Blocks: 8           IO
4     Block: 4096  regular file
5 Device: 8,1 Inode: 2622512      Links: 1
6 Access: (0644/-rw-r--r--)  Uid: (    0/
7   root)  Gid: (    0/  root)
8 Access: 2024-05-21 18:30:27.000000000 +0530
9 Modify: 2024-04-07 23:32:30.000000000 +0530
10 Change: 2024-05-21 18:30:27.047718323 +0530
11 Birth: 2024-05-21 18:30:27.047718323 +0530
  
```

We can also specify the format of the output of the `stat` command using the `--format` or `-c` flag to print only the metadata we want.

The data of the file is stored in the storage block. The inode number indexes a table of inodes on the file system. From the inode number, the kernel's file system driver can access the inode contents, including the location of the file, thereby allowing access to the file.

On many older file systems, inodes are stored in one or more fixed-size areas that are set up at file system creation time, so the maximum number of inodes is fixed at file system creation, limiting the maximum number of files the file system can hold.

Metadata	Description
Size	Size of the file in bytes
Blocks	Number of blocks used by the file
IO Block	Block size of the file system
Device	Device ID of the file system
Inode	Inode number of the file
Links	Number of hard links to the file
Access	Access time of the file ( <code>atime</code> )
Modify	Modification time of the file ( <code>mtime</code> )
Change	Change time of the inode ( <code>ctime</code> )
Birth	Creation time of the file

**Table 1.11:** Metadata of a File

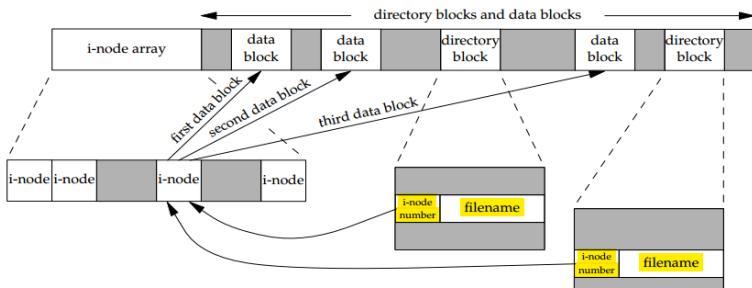
Some Unix-style file systems such as JFS, XFS, ZFS, OpenZFS, ReiserFS, btrfs, and APFS omit a fixed-size inode table, but must store equivalent data in order to provide equivalent capabilities. Common alternatives to the fixed-size table include B-trees and the derived B+ trees.

**Remark 1.6.2** Although the inodes store the metadata of the file, the filename is not stored in the inode. It is stored in the directory entry. Thus the filename, file metadata, and file data are stored separately.

### 1.6.3 Directory Entries

Unix directories are lists of association structures, each of which contains one filename and one inode number. The file system driver must search a directory for a particular filename and then convert the filename to the correct corresponding inode number.

Thus to read a file from a directory, first the directory's directory entry is read which stores the name



**Figure 1.11:** Inodes and Directory Entry

of the file and its inode number. The kernel then follows the inode number to find the inode of the file. The inode stores all the metadata of the file, and the location of the data of the file. The kernel then follows the inode to find the data of the file. This is shown in Figure 1.11.

So what happens if two directory entries point to the same inode? This is called a **hard link**.

#### 1.6.4 Hard Links

If multiple entries of the directory entry points to the same inode, they are called hard links. Hard links can have different names, but they are the same file. As they point to the same inode, they also have the same metadata.

This is useful if you want to have the same file in multiple directories without taking up more space. It is also useful if you want to keep a backup of an important file which is accessed by many people. If someone accidentally deletes the file, the other hard links will still be there and able to access the file.

**Definition 1.6.2 (Hard Links)** Hard Links are just pointers to the same inode. They are the same file. They are not pointers to the path of the file. They are pointers to the file itself. They are not affected by the deletion of the other file. When creating a hard link, you need to provide the path of the original file, and thus it has to be either absolute path, or relative from the current working directory, not relative from the location of the hard link.

Hard links can be created for files only, and not directories. It can be created using the `ln` command.

```
1 | $ ln file1 file2
```

This will create a hard link named `file2` that points to the same inode as `file1`.

**Remark 1.6.3** Hard links are not dependent on the original file. They are the same file and equivalent. The first link to be created has no special status.

Historically directories could also have hard links, but this would cause the file tree to stop being a Directed Acyclic Graph<sup>50</sup> and become a Directed Cyclic Graph if a hardlink of an ancestor was put as a subdirectory. This would create confusions and infinite walks in the file system. Modern systems generally prohibit this confusing state, except that the parent of root is still defined as root.<sup>51</sup>

As hard links depend on the inode, they can only exist in the same filesystem as inodes are unique to a filesystem only.

If we want to create shortcuts across filesystems, or

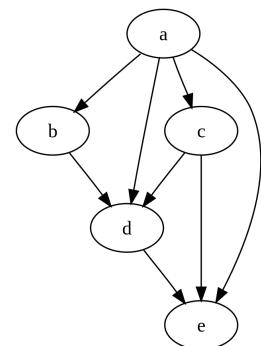


Figure 1.12: Directed Acyclic Graph

50: A Directed Acyclic Graph is a graph that has no cycles as seen in Figure 1.12.

51: The most notable exception to this prohibition is found in Mac OS X (versions 10.5 and higher) which allows hard links of directories to be created by the superuser.

if we want to create a link to a directory, we can use **symbolic links**.

### 1.6.5 Symbolic Links

A symbolic link contains a text string that is automatically interpreted and followed by the operating system as a path to another file or directory. This other file or directory is called the "target". The symbolic link is a second file that exists independently of its target. If a symbolic link is deleted, its target remains unaffected. If a symbolic link points to a target, and sometime later that target is moved, renamed or deleted, the symbolic link is not automatically updated or deleted, but continues to exist and still points to the old target, now a non-existing location or file. Symbolic links pointing to moved or non-existing targets are sometimes called broken, orphaned, dead, or dangling.

**Definition 1.6.3 (Soft Links)** Soft Links are special kinds of files that just store the path given to them. Thus the path given while making soft links should either be an absolute path, or relative **from** the location of the soft link **to** the location of the original file. It should not be relative from current working directory.<sup>a</sup>

<sup>a</sup> This is a common mistake.

Symlinks are created using the `symlink` system call. This can be done using the `ln -s` command.

```
1 | $ echo "hello" > file1
2 | $ ln -s file1 file2
3 | $ ls -l
4 | total 4
```

```

5 -rw-r--r-- 1 sayan sayan 6 May 23 15:27 file1
6 lrwxrwxrwx 1 sayan sayan 5 May 23 15:27 file2
    -> file1
7 $ cat file2
8 hello

```

### Interesting Observation:

Usually we have seen that if we use `ls -l` with a directory as its argument, it lists the contents of the directory.

The only way to list the directory itself is to use `ls -ld`.

But if a symlink is made to a directory, then `ls -l` on that symlink will list only the symlink.

To list the contents of the symlinked directory we have to append a / to the symlink.

```

1 $ ln -s /etc /tmp/etc
2 $ ls -l /tmp/etc
3 lrwxrwxrwx 1 sayan sayan 4 May 23 15:30 /tmp/
    etc -> /etc
4 $ ls -l /tmp/etc/ | head -n5
5 total 1956
6 -rw-r--r-- 1 root root      44 Mar 18 21:50
    adjtime
7 drwxr-xr-x 3 root root   4096 Nov 17 2023
    als
8 -rw-r--r-- 1 root root     541 Apr  8 20:53
    anacrontab
9 drwxr-xr-x 4 root root   4096 May 19 00:44
    apparmor.d

```

Here I used `head` to limit the number of lines shown as the directory is large.<sup>52</sup>

The symlink file stores only the path provided to it while creating it. This was historically stored in the data block which was pointed to by the inode. But this made it slower to access the symlink.

<sup>52</sup>: This way of combining commands will be discussed later.

Modern systems store the symlink value in the inode itself if its not too large. Inodes usually have a limited space allocated for each of them, so a symlink with a small target path is stored directly in the inode. This is called a **fast symlink**.

However if the target path is too large, it is stored in the data block pointed to by the inode. This is retroactively called a **slow symlink**.

This act of storing the target path in the inode is called **Inlining**.

Symlinks do not have a permission set, thus they always report `lrwxrwxrwx` as their permissions.

The size reported of a symlink file is independent of the actual file's size.

```

1 $ echo "hello" > file1
2 $ ln -s file1 file2
3 $ ls -l
4 -rw-r--r-- 1 sayan sayan 6 May 23 15:27 file1
5 lrwxrwxrwx 1 sayan sayan 5 May 23 15:27 file2
       -> file1
6 $ echo "a very big file" > file2
7 $ ls -l
8 -rw-r--r-- 1 sayan sayan 16 May 23 15:40 file1
9 lrwxrwxrwx 1 sayan sayan 5 May 23 15:27 file2
       -> file1

```

Rather, the size of a symlink is the length of the target path.

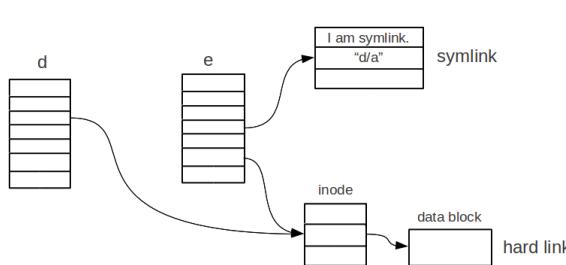
```

1 $ ln -s /a/very/long/and/non-existant/path
      link1
2 $ ln -s small link2
3 $ ls -l
4 total 0
5 lrwxrwxrwx 1 sayan sayan 34 May 23 15:41 link1
       -> /a/very/long/and/non-existant/path
6 lrwxrwxrwx 1 sayan sayan  5 May 23 15:41 link2
       -> small

```

Notice that the size of `link1` is 34, the length of the target path, and the size of `link2` is 5, the length of the target path.

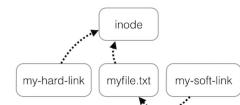
### 1.6.6 Symlink vs Hard Links



The difference between a symlink and a hard link is that a symlink is a pointer to the original file, while a hard link is the same file. Other differences are listed in Table 1.12.

**Table 1.12:** Symlink vs Hard Link

Property	Symlink	Hard Link
File Type	Special File	Regular File
Size	Length of the target path	Size of the file
Permissions	<code>lrwxrwxrwx</code>	Same as the original file
Inode	Different	Same
Dependency	Dependent on the original file across filesystems	Independent of the original file in the same filesystem
Creation	across filesystems	
Target	Can point to directories	Can only point to files



**Figure 1.13:** Abstract Representation of Symbolic Links and Hard Links

**Figure 1.14:** Symbolic Links and Hard Links

### 1.6.7 Identifying Links

#### Soft Links:

To identify if a file is a symlink or a hard link, you can use the `ls -l` command. If the file is a symlink,

the first character of the permissions will be `l`. `ls -l` will also show the target of the symlink after a `->` symbol. However, you cannot ascertain if a file has a soft link pointing to it somewhere else or not.

### Hard Links:

To identify if a file is a hard link, you can use the `ls -i` command. Hard links will have the same inode number as each other. The inode number is the first column of the output of `ls -i`.

53: third if using `ls -li`

Also the number of links to the file will be more than 1. The number of links is the second<sup>53</sup> column of the output of `ls -l`.

Even if a hard link is not present in current directory, you can ascertain that a file has a hard link pointing to it somewhere else using the **number of hardlinks** column of `ls -l`.

```

1 $ touch file1
2 $ ln -s file1 file2
3 $ ln file1 file3
4 $ ls -li
5 total 0
6 4850335 -rw-r--r-- 2 sayan sayan 0 May 23
    15:56 file1
7 4851092 lrwxrwxrwx 1 sayan sayan 5 May 23
    15:56 file2 -> file1
8 4850335 -rw-r--r-- 2 sayan sayan 0 May 23
    15:56 file3

```

### 1.6.8 What are . and ..?

`.` and `..` are special directory entries. They are hard links to the current directory and the parent directory respectively. Each directory has a `.` entry pointing to itself and a `..` entry pointing to its parent directory.

Due to this, the number of hard links to a directory is exactly equal to the number of subdirectories it has plus 2.

This is because each subdirectory has a .. entry pointing to the parent directory, and the parent directory has a . entry pointing to itself.

So the directory's name in its parent directory is 1 link, the . in the directory is 1 link, and all the subdirectories have a .. entry pointing to the directory, which is 1 link each.

$$\text{Number of links to a directory} = \text{Number of subdirectories} + 2$$

This formula always stands because a user cannot create additional hard links to a directory.

#### **Question 1.6.1** How to list the inodes of a file?

**Answer 1.6.1** `ls -i` will list the inodes of a file. The inodes are the first column of the output of `ls -i`. This can be combined with other flags like `-l` or `-a` to show more details.

#### **Question 1.6.2** How to create soft link of a file?

**Answer 1.6.2** `ln -s sourcefile targetfile` will create a soft link of `sourcefile` named `targetfile`. The soft link is a pointer to the original file.

#### **Question 1.6.3** How to create hard link of a file?

**Answer 1.6.3** `ln sourcefile targetfile` will create a hard link of `sourcefile` named `targetfile`. The hard link is same as the original file. It does not depend on the original file anymore after creation. They are equals, both are hardlinks of each other. There is no parent-child relationship. The other file can be deleted and the original file will still work.

**Question 1.6.4** How to get the real path of a file?

Assume three files:

- ▶ `file1` is a soft link to `file2`
- ▶ `file2` is a soft link to `file3`
- ▶ `file3` is a regular file

Real path of all these three should be the same.  
How to get that?

**Answer 1.6.4** `realpath filename` will give the real path of `filename`.

You can also use `readlink -f filename` to get the real path.

# Command Line Editors

## 2.1 Introduction

Now that we know how to go about navigating the linux based operating systems, we might want to view and edit files. This is where command line editors come in.

**Definition 2.1.1** A **command line editor** is a type of text editor that operates entirely from the command line interface. They usually do not require a graphical user interface or a mouse.<sup>a</sup>

<sup>a</sup> This means that CLI editors are the only way to edit files when you are connected to a remote server. Since remote servers do not have any graphical server like X11, you cannot use graphical editors like gedit or kate.

### 2.1.1 Types of Editors

- ▶ **Graphical Editors:** These are editors that require a graphical user interface. Examples include *gedit*<sup>\*</sup>, *kate*<sup>†</sup>, *vs code*, etc.
- ▶ **Command Line Editors:** These are editors that operate entirely from the command line interface.

We will be only discussing command line editors in this chapter.

<sup>\*</sup> Gedit is the default text editor in GNOME desktop environment.

<sup>†</sup> Kate is the default text editor in KDE desktop environment.

## 2.1.2 Why Command Line Editors?

1: SSH stands for Secure Shell. It is a cryptographic network protocol for operating network services securely over an unsecured network. This will be discussed in detail in a later chapter.

Command line editors are very powerful and efficient. They let you edit files without having to leave the terminal. This is usually faster than opening a graphical editor. In many cases like sshing<sup>1</sup> into a remote server, command line editors are the only way to edit files. Another reason for the popularity of command line editors is that they are very lightweight.

## 2.1.3 Mouse Support

Most command line editors do not have mouse support, and others do not encourage it. But won't it be difficult to navigate without a mouse? Not really. Once you get used to the keyboard shortcuts, you will find that you can navigate way faster than with a mouse.

Mouse editors usually require the user to click on certain buttons, or to follow multi-click procedures in nested menus to perform certain tasks.

Whereas in keyboard based editors, all the actions that can be performed are mapped to some keyboard shortcuts.

Modern CLI editors usually also allow the user to totally customize the keyboard shortcuts.

2: X11 and Wayland are display servers that are used to render graphical applications. Although not directly covered, these can be explored in details on the internet.

This being said, most modern CLI editors do have mouse support as well if the user is running them in a terminal emulator that supports mouse over a X11 or Wayland display server.<sup>2</sup>

### 2.1.4 Editor war

Although there are many command line editors available, the most popular ones are *vim* and *emacs*.

**Definition 2.1.2** The editor war is the rivalry between users of the Emacs and vi (now usually Vim, or more recently Neovim) text editors. The rivalry has become an enduring part of hacker culture and the free software community.<sup>3</sup>

3: More on this including the history and the humor can be found on the [internet](#).

#### Vim

Vim is a modal editor, meaning that it has different modes for different tasks. Most editors are modeless, this makes vim a bit difficult to learn. However, once familiar with it, it is very powerful and efficient. Vim heavily relies on alphanumeric keys for navigation and editing. Vim keybindings are so popular that many other editors and even some browsers<sup>4</sup> have vim-like keybindings.

#### Emacs

Emacs is a modeless editor, meaning that it does not have different modes for different tasks. Emacs is also very powerful and efficient. It uses multi-key combinations for navigation and editing.

4: *qutebrowser* is a browser that uses vim-like keybindings. Firefox and Chromium based browsers also have extensions that provide vim-like keybindings. These allow the user to navigate the browser using vim-like keybindings and without ever touching the mouse.

### 2.1.5 Differences between Vim and Emacs

#### Keystroke execution

Emacs commands are key combinations for which modifier keys are held down while other keys are pressed; a command gets executed once completely typed.

Vim retains each permutation of typed keys (e.g. order matters). This creates a path in the decision tree which unambiguously identifies any command.

### **Memory usage and customizability**

Emacs executes many actions on startup, many of which may execute arbitrary user code. This makes Emacs take longer to start up (even compared to vim) and require more memory. However, it is highly customizable and includes a large number of features, as it is essentially an execution environment for a Lisp program designed for text-editing.

Vi is a smaller and faster program, but with less capacity for customization. vim has evolved from vi to provide significantly more functionality and customization than vi, making it comparable to Emacs.

### **User environment**

Emacs, while also initially designed for use on a console, had X11 GUI support added in Emacs 18, and made the default in version 19. Current Emacs GUIs include full support for proportional spacing and font-size variation. Emacs also supports embedded images and hypertext.

Vi, like emacs, was originally exclusively used inside of a text-mode console, offering no graphical user interface (GUI). Many modern vi derivatives, e.g. MacVim and gVim, include GUIs. However, support for proportionally spaced fonts remains absent. Also lacking is support for different sized fonts in the same document.

### **Function/navigation interface**

Emacs uses metakey chords. Keys or key chords can be defined as prefix keys, which put Emacs into

a mode where it waits for additional key presses that constitute a key binding. Key bindings can be mode-specific, further customizing the interaction style. Emacs provides a command line accessed by M-x that can be configured to autocomplete in various ways. Emacs also provides a defalias macro, allowing alternate names for commands.

Vi uses distinct editing modes. Under "insert mode", keys insert characters into the document. Under "normal mode" (also known as "command mode", not to be confused with "command-line mode", which allows the user to enter commands), bare keypresses execute vi commands.

## Keyboard

The expansion of one of Emacs' backronyms is Escape, Meta, Alt, Control, Shift, which neatly summarizes most of the modifier keys it uses, only leaving out Super. Emacs was developed on Spacecadet keyboards that had more key modifiers than modern layouts. There are multiple emacs packages, such as spacemacs or ergoemacs that replace these key combinations with ones easier to type, or customization can be done ad hoc by the user.

Vi does not use the Alt key and seldom uses the Ctrl key. vi's keyset is mainly restricted to the alphanumeric keys, and the escape key. This is an enduring relic of its teletype heritage, but has the effect of making most of vi's functionality accessible without frequent awkward finger reaches.

## Language and script support

Emacs has full support for all Unicode-compatible writing systems and allows multiple scripts to be freely intermixed.

Vi has rudimentary support for languages other than English. Modern Vim supports Unicode if used with a terminal that supports Unicode.

### 2.1.6 Nano: The peacemaker amidst the editor war

*Nano* is a simple command line editor that is easy to use. It does not have the steep learning curve of vim or emacs. But it is not as powerful as vim or emacs as well. It is a common choice for beginners who just want to append a few lines to a file or make a few changes. It is also a non-modal editor like editor which uses modifier chording like emacs. However, it mostly uses the control key for this purpose and has only simple keybindings such as *Ctrl+O* to save and *Ctrl+X* to exit.

## 2.2 Vim

### 2.2.1 History

The history of Vim is a very long and interesting one.

#### Teletypes

**Definition 2.2.1** A **teletype** (TTY) or a teleprinter is a device that can send and receive typed messages from a distance.

**Table 2.1:** History of Vim

		QED text editor by Butler Lampson and Peter Deutsch for Berkeley Timesharing System.
1967	.....	Ken Thompson and Dennis Ritchie's QED for MIT CTSS, Multics, and GE-TSS.
1969	.....	Ken Thompson releases ed - The Standard Text Editor.
1976	.....	George Coulouris and Patrick Mullaney release em - The Editor for Mortals.
1976	.....	Bill Joy and Chuck Haley build upon em to make en, which later becomes ex.
1977	.....	Bill Joy adds visual mode to ex.
1979	.....	Bill Joy creates a hardlink 'vi' for ex's visual mode.
1987	.....	Tim Thompson develops a vi clone for the Atari ST named STevie (ST editor for VI enthusiasts).
1988	.....	Bram Moolenaar makes a stevie clone for the Amiga named Vim (Vi IMitation).

Very early computers used to use teletypes as the output device. These were devices that used ink and paper to actually *print* the output of the computer.



Figure 2.1: A Teletype

These did not have an *automatic* refresh rate like modern monitors. Only when the computer sent a signal to the teletype, would the teletype print the output.

Due to these restrictions it was not economical or practical to print the entire file on the screen. Thus most editors used to print only one line at a time on the screen and did not have updating graphics.



Figure 2.2: Ken Thompson

## QED

QED was a text editor developed by Butler Lampson and Peter Deutsch in 1967 for the Berkeley Time-sharing System. It was a character-oriented editor that was used to create and edit text files. It used to print or edit only one character at a time on the screen. This is because the computers at that time used to use a teletype machine as the output device, and not a monitor.

Ken Thompson used this QED at Berkeley before he came to Bell Labs, and among the first things he did on arriving was to write a new version for the MIT CTSS system. Written in IBM 7090 assembly language, it differed from the Berkeley version most notably in introducing regular expressions<sup>5</sup> for specifying strings to seek within the document being edited, and to specify a substring for which a substitution should be made. Until that time, text editors could search for a literal string, and substitute for one, but not specify more general strings.

Ken not only introduced a new idea, he found an inventive implementation: on-the-fly compiling. Ken's QED compiled machine code for each regular expression that created a NDFA (non-deterministic finite automaton) to do the search. He published

- 5: Regular expressions are a sequence of characters that define a search pattern. Usually this pattern is used by string searching algorithms for "find" or "find and replace" operations on strings. This will be discussed in detail in a later chapter.

this in C. ACM 11 #6, and also received a patent for the technique: US Patent #3568156.

While the Berkeley QED was character-oriented, the CTSS version was line-oriented. Ken's CTSS qed adopted from the Berkeley one the notion of multiple buffers to edit several files simultaneously and to move and copy text among them, and also the idea of executing a given buffer as editor commands, thus providing programmability.

When developing the MULTICS project, Ken Thompson wrote yet another version of QED for that system, now in BCPL<sup>6</sup> and now created trees for regular expressions instead of compiling to machine language.

In 1967 when Dennis Ritchie joined the project, Bell Labs had slowly started to move away from Multics. While he was developing the initial stages of Unix, he rewrote QED yet again, this time for the GE-TSS system in Assembly language. This was well documented, and was originally intended to be published as a paper.<sup>7</sup>

## ED

After their experience with multiple implementations of QED, Ken Thompson wrote **ed** in 1969.

This was now written in the newly developed B language, a predecessor to C. This implementation was much simpler than QED, and was line oriented. It stripped out much of regular expression support, and only had the support for \*. It also got rid of multiple buffers and executing contents of buffer.

Slowly, with time, Dennis Ritchie created the C language, which is widely in use even today.

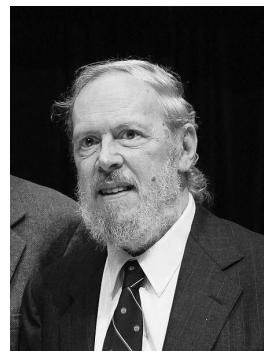


Figure 2.3: Dennis Ritchie

6: BCPL ("Basic Combined Programming Language") is a procedural, imperative, and structured programming language. Originally intended for writing compilers for other languages, BCPL is no longer in common use.

7: At that time, systems did not have a standardized CPU architecture or a generalized low level compiler. Due to this, applications were not portable across systems. Each machine needed its own version of the application to be written from the scratch, mostly in assembly language.

The reference manual for GE-TSS QED can still be found on [Dennis Ritchie's website](#). Much of this information is taken from his [blog](#).

Ken Thompson re-wrote **ed** in C, and added back some of the complex features of QED, like back references in regular expressions.

Ed ended up being the **Standard Text Editor** for Unix systems.

**Remark 2.2.1** Since all of Bell-Labs and AT&T's software was proprietary, the source code for *ed* was not available to the public. Thus, the *ed* editor accessible today in **GNU/Linux**, is another implementation of the original *ed* editor by the **GNU project**.

However, **ed** was not very user friendly and it was very terse. Although this was originally intented, since it would be very slow to print a lot of diagnostic messages on a teletype, slowly, as people moved to faster computers and monitors, they wanted a more user friendly editor.

## VDU Terminals



**Figure 2.4:** Xerox Alto, one of the first VDU terminals with a GUI, released in 1973

**Definition 2.2.2** A terminal that uses video display technology like cathode ray tubes (CRT) or liquid crystal displays (LCD) to display the terminal output is called a **VDU terminal**. (Video Display Unit)

These terminals were able to show video output, instead of just printing the output on paper. Although initially these were very expensive, and were not a household item, they were present in the research parks like Xerox PARC.

## EM

George Coulouris (not the actor) was one of the people who had access to these terminals in his work at the Queen Mary College in London.

The drawbacks of **ed** were very apparent to him when using on these machines.

He found that the UNIX's **raw mode**, which was at that time totally unused, could be used to give some of the convenience and immediacy of feedback for text editing.

He claimed that although the **ed** editor was groundbreaking in its time, it was not very user friendly. He termed it as not being an editor for mortals.

He thus wrote **em** in 1976, which was an **Editor for Mortals**.<sup>8</sup>

Although em added a lot of features to ed, it was still a line editor, that is, you could only see one line at a time. The difference from ed was that it allowed visual editing, meaning you can see the state of the line as you are editing it.

Whereas most of the development of Multics and Unix was done in the United States, the development of **em** was done in the United Kingdom, in the Queen Mary College, which was the first college in the UK to have UNIX.

## EN

In the summer of 1976, George Coulouris was a visiting professor at the University of California, Berkeley. With him, he had brought a copy of his **em** editor on a Dectape<sup>9</sup> and had installed it there on their departmental computers which were still using teletype terminals. Although em was designed for VDU terminals, it was still able to run (albeit slowly) on the teletype terminals by printing the current line every time.

<sup>8:</sup> George named em as Editor for Mortals because Ken Thompson visited his lab at QMC while he was developing it and said something like: "yeah, I've seen editors like that, but I don't feel a need for them, I don't want to see the state of the file when I'm editing". This made George think that Ken was not a mortal, and thus he named it Editor for Mortals.



Figure 2.5: A first generation Dectape (bottom right corner, white round tape) being used with a PDP-11 computer

<sup>9:</sup> A Dectape is a magnetic tape storage device that was used in the 1970s. It was used to store data and programs.



Figure 2.6: George Coulouris

There he met Bill Joy, who was a PhD student at Berkeley. On showing him the editor, Bill Joy was very impressed and wanted to use it on the PDP-11 computers at Berkeley. The system support team at Berkley were using PDP-11 which used VDU Terminals, an environment where em would really shine.

He explained that 'em' was an extension of 'ed' that gave key-stroke level interaction for editing within a single line, displaying the up-to-date line on the screen (a sort of single-line screen editor). This was achieved by setting the terminal mode to 'raw' so that single characters could be read as they were typed - an eccentric thing for a program to do in 1976.

Although the system support team at Berkeley were impressed by this editor, they knew that if this was made available to the general public, it would take up too much resources by going to the raw mode on every keypress. But Bill and the team took a copy of the source code just to see if they might use it.

George then took a vacation for a few weeks, but when he returned, he found that Bill had taken his ex as a starting point and had added a lot of features to it. He called it **en** initially, which later became **ex**.

## EX

Bill Joy took inspiration from several other ed clones as well, and their own tweaks to ed, although the primary inspiration was **em**. Bill and Chuck Haley built upon **em** to make **en**, which later became **ex**.

This editor had a lot of improvements over **em**, such as adding the ability to add abbreviations (using the **ab** command), and adding keybindings (maps).



Figure 2.7: Bill Joy

It also added the ability to mark some line using the `k` key followed by any letter, and then jump to that line from any arbitrary line using the `'` key followed by the letter.

Slowly, with time, the modern systems were able able to handle the raw mode, and real time editing more and more. This led to the natural progression, What if we could see the entire file at once, and not just one line at a time?

## VI

Bill added the **visual mode** to ex in 1977.<sup>10</sup> This was not a separate editor, but rather just another mode of the `ex` editor. You could open `ex` in visual mode using the `-v` flag to `ex`.

```
1 | $ ex -v filename
```

This visual mode was the first time a text editor was **modal**. This means that the editor had different modes for different tasks. When you want to edit text, you would go to the insert mode, and type the text. When you want to navigate, you would go to the normal mode, and use the navigation keys and other motions defined in `vi`.

Slowly, as the visual mode became more and more popular, Bill added a hardlink to `ex` called **vi**.<sup>11</sup>

The modal version of `vi` was also inspired from another editor called **bravo**, which was developed at Xerox PARC.<sup>12</sup>

If you use `vi/vim`, you may notice that the key to exit the insert mode is `Esc`. This may seem inconveniently placed at the top left corner, but this was because the original `vi` was developed on a ADM-3A terminal, which had the `Esc` key to the left of the `Q` key, where modern keyboards have the `Tab` key.<sup>13</sup>

<sup>10</sup>: This visual mode is not the same as the visual mode in `vim`.

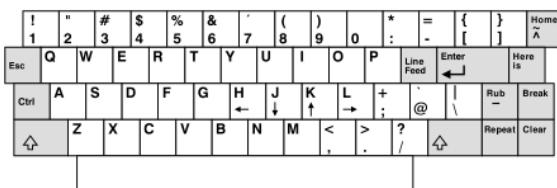
<sup>11</sup>: This means that it did not take up additional space on the disk, but was just another entry in the directory entry that pointed to the same inode which stored the `ex` binary path. Upon execution, `ex` would detect if it was called as `vi` and would start in visual mode by default. We have covered hardlinks in Chapter 1.

<sup>12</sup>: Xerox PARC has always been ahead of its time. The first graphical user interface was developed at Xerox PARC. The bravo editor used bitmapped graphics to display the text, and had extensive mouse support. The overdependence on the mouse in such an early time was one of the reasons that the bravo editor was not as popular as `vi`.

<sup>13</sup>: Since the placement of the Escape key is inconvenient in modern keyboard layouts, many people remap the Escape key to the Caps Lock key either in `vim` or in the operating system itself.

Also, the choice of h,j,k,l for navigation was because the ADM-3A terminal did not have arrow keys, rather, it had h,j,k,l keys for navigation.

This can be seen in Figure 2.8.



**Figure 2.8:** The Keyboard layout of the ADM-3A terminal

Bill Joy was also one of the people working on the **Berkeley Software Distribution (BSD) of Unix**. Thus he bundled vi with the first BSD distribution of UNIX released in 1978. The pre-installed nature of vi in the BSD Distribution made it very popular.

However, since both the source code of ed was restricted by Bell Labs - AT&T, and the source code of vi was restricted by the University of California, Berkeley, they could not be modified by the users or distributed freely.

This gave birth to a lot of clones of vi.

### Vi Clones

The vi clones were written because the source code for the original version was not freely available until recently. This made it impossible to extend the functionality of vi. It also precluded porting vi to other operating systems, including Linux.

- **calvin:** a freeware "partial clone of vi" for use on MS-DOS. It has the advantages of small size (the .exe file is only 46.1KB!) and fast execution but the disadvantage that it lacks many of the ex commands, such as search and replace.



**Figure 2.9:** Stevie Editor

- ▶ **lemmy:** a shareware version of vi implemented for the Microsoft Windows platforms which combines the interface of vi with the look and feel of a Windows application.
- ▶ **nvi:** It is a re-implementation of the classic Berkeley vi editor, derived from the original 4.4BSD version of vi. It is the "official" Berkeley clone of vi, and it is included in FreeBSD and the other BSD variants.
- ▶ **stevie:** 'ST Editor for VI Enthusiasts' was developed by Tim Thompson for the Atari ST. It is a clone of vi that runs on the Atari ST. Tim Thompson wrote the code from scratch (not based on vi) and posted its source code as a free software to [comp.sys.atari.st](#) on June 1987. Later it was ported to UNIX, OS/2, and Amiga. Because of this independence from vi and ed's closed source license, most vi clones would base their work off of stevie to keep it free and open source.
- ▶ **elvis:** Elvis creator, Steve Kirkendall, started thinking of writing his own editor after Stevie crashed on him, causing him to lose hours of work and damaging his confidence in the editor. Stevie stored the edit buffer in RAM, which Kirkendall believed to be impractical on the MINIX operating system. One of Kirkendall's main motivation for writing his own vi clone was that his new editor stored the edit buffer in a file instead of storing it in RAM. Therefore, even if his editor crashed, the edited text could still be retrieved from that external file. Elvis was one of the first vi clones to offer support for GUI and syntax highlighting.

The clones add numerous new features which make them significantly easier to use than the original

vi, especially for neophytes. A particularly useful feature in many of them is the ability to edit files in multiple windows. This facilitates working on more than one file at the same time, including cutting and pasting text among them.

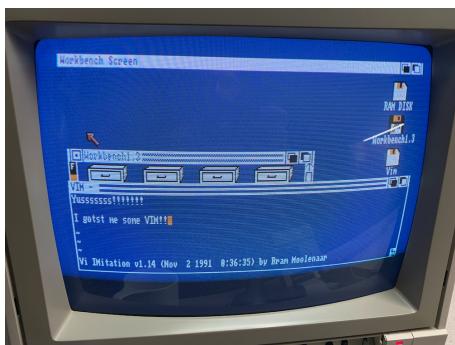
Many of the clones also offer GUI versions of vi that operate under the X Windows system and can take advantage of bit-mapped (high resolution) displays and the mouse.

### Vim

Bram Moolenaar, a Dutch programmer, was impressed by STeVIE, a vi clone for the Atari ST. But he was working with the Commodore Amiga at that time, and there was no vi clone for the Amiga. So Bram began working on the stevie clone for the AmigaOS in 1988.



**Figure 2.10:** Bram Moolenaar



**Figure 2.11:** The initial version of Vim, when it was called Vi IMitation

He released the first public release (v 1.14) in 1991 as visible in Figure 2.11.

Since Vim was based off of Stevie, and not ed or vi so it could be freely distributed. It was licensed under a charityware license, named as Vim License. The license stated that if you liked the software, you should consider making a donation to a charity of your choice.

Moolenaar was an advocate of a NGO based in Kibaale, Uganda, which he founded to support children whose parents have died of AIDS. In 1994, he volunteered as a water and sanitation engineer for the Kibaale Children's Centre and made several return trips over the following twenty-five years.

Later Vim was re-branded as 'Vi IMproved' as seen in Figure 2.12.

Vim has been in development for over 30 years now, and is still actively maintained. It has added a lot of features over the years, such as syntax highlighting, plugins, completion, PCRE support, mouse support, etc.

### **neovim**

Recently there have been efforts to modernize the vim codebase. Since it is more than 30 years old, it has a lot of legacy code. The scripting language of vim is also not a standard programming language, but rather a custom language called vimscript.

To counter this, a new project called **neovim** has been started. It uses **lua** as the scripting language, and has a lot of modern features like out of the box support for LSP,<sup>14</sup> better mouse integration, etc.

In this course, we will be learning only about basic vi commands and we will be using vim as the editor.



**Figure 2.12:** Vim 9.0 Start screen

14: LSP stands for Language Server Protocol. It is a protocol that allows the editor to communicate with a language server to provide features like autocompletion, go to definition, etc. This makes vim more like an IDE. This book is written using neovim.

The screenshot shows a Neo Vim window with two panes. The left pane displays a file tree of a LaTeX project named 'sc-handbook'. The right pane shows a help buffer for the 'textbf' command in Neovim. The help text discusses the evolution of Vim's scripting language, Vimscript, and its replacement by TextBuffer (textbf) in Neovim. It also covers LSP integration and other modern features.

```

7-editors.tex * 2 chars, 65 lines
+--- .git
|   +--- chapters
|   |   +--- chapter1.tex
|   |   +--- chapter2.tex
|   |   +--- chapter3.tex
|   |   +--- chapter4.tex
|   |   +--- chapter5.tex
|   |   +--- chapter6.tex
|   |   +--- chapter7.tex
|   +--- 10-sed.tex
|   +--- 11-awk.tex
|   +--- 12-hardware.tex
|   +--- 13-etc.tex
|   +--- 14-networking.tex
|   +--- 15-storage.tex
|   +--- 16-docker.tex
|   +--- 17-procman.tex
|   +--- 4-redirection.tex
|   +--- 5-variables.tex
|   +--- 6-regex.tex
|   +--- 7-editors.tex
|   +--- 8-commands.tex
|   +--- 9-scripts.tex
+--- chaps
|   +--- week1.tex
|   +--- week1.pdf
+--- images
|   +--- gitignore
|   +--- compileall.sh
|   +--- exts
|   +--- necessary.tex
|   +--- kee.sty
|   +--- keebook.sty
|   +--- keebook.cls
|   +--- keeerrors.sty
|   +--- keetheorems.sty
|   +--- README.ad
+--- main.tex
+--- main.pdf
+--- README.ad
+--- scratch.ch

M_NORMA NORMAL 7-editors.tex [master 0:651
1 projects/sc-handbook 2 bash
▲ 102 % LSP - TexLab 86-handbook 99 %
3 └─ unmatched "end of file /tmp/nvim.sayan/ZrHyR0/1151/7-editors.tex"

```

**Figure 2.13:** Neo Vim Window showing editing this book

## 2.2.2 Ed Commands

Before we move on to Vi Commands, let us first learn about the basic ed commands. These will also be useful in vim, since the ex mode of vim is based on ed/ex where we can directly use ed commands on our file in the buffer.

**Table 2.2:** Ed Commands

Description	Commands
Show the Prompt	P
Command Format	[addr[,addr]]cmd[params]
Commands for location	1 . \$ \%+- , ; /RE/
Commands for editing	f p a c d i j s m u
Execute a shell <i>command</i>	!command
edit a file	e filename
read file contents into buffer	r filename
read <i>command</i> output into buffer	r !command
write buffer to filename	w filename
quit	q

### Commands for location

**Table 2.3:** Commands for location

Commands	Description
a number like 2	refers to second line of file
.	refers to current line
\$	refers to last line
%	refers to all the lines
+	line after the cursor (current line)
-	line before the cursor (current line)
,	refers to buffer holding the file or last line in buffer
;	refers to current position to end of the file
/RE/	refers line matched by pattern specified by 'RE'

## Commands for Editing

Commands	Description
f	show name of file being edited
p	print the current line
a	append at the current line
c	change the current line
d	delete the current line
i	insert line at the current position
j	join lines
s	search for regex pattern
m	move current line to position
u	undo latest change

**Table 2.4:** Commands for Editing

Let us try out some of these commands in the ed editor.

Lets start with creating a file, which we will then open in the ed editor.

```

1 $ echo "line-1 hello world
2 line-2 welcome to line editor
3 line-3 ed is perhaps the oldest editor out
   there
4 line-4 end of file" > test.txt

```

This creates a file in the current working directory

with the name `test.txt` and the contents as given above.

We invoke ed by using the executable `ed` and providing the filename as an argument.

```
1 | $ ed test.txt
2 | 117
```

As soon as we run it, you will see a number, which is the number of characters in the file. The terminal may seem hung, since there is no prompt, of either the bash shell, or of the ed editor. This is because `ed` is a line editor, and it does not print the contents of the file on the screen.

Off the bat, we can observe the terseness of the `ed` editor since it does not even print a prompt. To turn it on, we can use the `P` command. The default prompt is `*`.

```
1 | ed test.txt
2 | 117
3 | P
4 | *
```

Now we can see the prompt `*` is always present, whenever the `ed` editor expects a command from the user.

Lets go to the first line of the file using the `1` command. We can also go to the last line of the file using the `$` command.

```
1 | *1
2 | line-1 hello world
3 | *$ 
4 | line-4 end of file
5 | *
```

To print out all the lines of the file, we can use the `,` or `\%` with `p` command.

```

1 *,p
2 line-1 hello world
3 line-2 welcome to line editor
4 line-3 ed is perhaps the oldest editor out
   there
5 line-4 end of file

1 *%p
2 line-1 hello world
3 line-2 welcome to line editor
4 line-3 ed is perhaps the oldest editor out
   there
5 line-4 end of file

```

However, if we use the , command without the p command, it will not print all the lines. Rather, it will just move the cursor to the last line and print the last line.

```

1 *,
2 line-4 end of file

```

We can also print any arbitrary line range using the line numbers separated by a comma and followed by the p command.

```

1 *2,3p
2 line-2 welcome to line editor
3 line-3 ed is perhaps the oldest editor out
   there

```

One of the pioneering features of ed was the ability to search for a pattern in the file. Let us quickly explain the syntax of the search command.<sup>15</sup>

```

1 */hello/
2 line-1 hello world

```

We may or may not include the p command after the last / in the search command.

We can advance to the next line using the + command.

<sup>15</sup>: The details of regular expressions will be covered in a later chapter.

```

1 *p
2 line-1 hello world
3 */
4 line-2 welcome to line editor

```

And go to the previous line using the - command.

```

1 *p
2 line-2 welcome to line editor
3 */
4 line-1 hello world

```

We can also print all the lines from the current line to the end of the file using the ;p command.

```

1 *.
2 line-2 welcome to line editor
3 *;p
4 line-2 welcome to line editor
5 line-3 ed is perhaps the oldest editor out
     there
6 line-4 end of file

```

We can also run arbitrary shell commands using the ! command.

```

1 *! date
2 Mon Jun 10 11:36:34 PM IST 2024
3 !

```

The output of the command is shown to the screen, however, it is not saved in the buffer.

To read the output of a command into the buffer, we can use the r command.

```

1 *r !date
2 32
3 *%p
4 line-1 hello world
5 line-2 welcome to line editor
6 line-3 ed is perhaps the oldest editor out
     there
7 line-4 end of file

```

8 | Mon Jun 10 11:37:42 PM IST 2024

The output after running the `r !date` command is the number of characters read into the buffer. We can then print the entire buffer using the `\%p` command.

The read data is appended to the end of the file.

We can write the buffer<sup>16</sup> to the disk using the `w` command.

```
1 | *w
2 | 149
```

The output of the `w` command is the number of characters written to the file.

To exit ed, we can use the `q` command.

```
1 | *q
```

To delete a line, we can use the `d` command. Lets say we do not want the date output in the file. We can re-open the file in ed and remove the last line.

```
1 | $ ed test.txt
2 | 149
3 | P
4 | *$*
5 | Mon Jun 10 11:38:49 PM IST 2024
6 | *d
7 | *%p
8 | line-1 hello world
9 | line-2 welcome to line editor
10 | line-3 ed is perhaps the oldest editor out
   |     there
11 | line-4 end of file
12 | *wq
13 | 117
```

16: Remember that the buffer is the in-memory copy of the file and any changes made to the buffer are not saved to the file until we write the buffer to the file.

We can add lines to the file using the `a` command. This appends the line after the current line. On entering this mode, the editor will keep on taking

input for as many lines as we want to add. To end the input, we can use the . command on a new line.

```

1 $ ed test.txt
2 117
3 P
4 *3
5 line-3 ed is perhaps the oldest editor out
   there
6 *a
7 perhaps not, since we know it was inspired
   from QED
8 which was made multiple times by thompson and
   ritchie
9 before ed was made.
10 .
11 *%p
12 line-1 hello world
13 line-2 welcome to line editor
14 line-3 ed is perhaps the oldest editor out
   there
15 perhaps not, since we know it was inspired
   from QED
16 which was made multiple times by thompson and
   ritchie
17 before ed was made.
18 line-4 end of file
19 *
```

We can also utilize the regular expression support in ed to perform search and replace operations. This lets us either search for a fixed string and replace with another fixed string, or search for a pattern and replace it with a fixed string.

Let us change hello world to hello universe.

```

1 *1
2 line-1 hello world
3 *s/world/universe/
4 line-1 hello universe
```

```

5 *%p
6 line-1 hello universe
7 line-2 welcome to line editor
8 line-3 ed is perhaps the oldest editor out
      there
9 perhaps not, since we know it was inspired
      from QED
10 which was made multiple times by thompson and
      ritchie
11 before ed was made.
12 line-4 end of file
13 *

```

We can print the name of the currently opened file using the f command.

```

1 *f
2 test.txt

```

If we wish to join two lines, we can use the j command. Let us join lines 4,5, and 6.

```

1 *4
2 perhaps not, since we know it was inspired
      from QED
3 *5
4 which was made multiple times by thompson and
      ritchie
5 *6
6 before ed was made.
7 *4,5j
8 *4
9 perhaps not, since we know it was inspired
      from QEDwhich was made multiple times by
      thompson and ritchie
10 *5
11 before ed was made.
12 *4,5j
13 *4
14 perhaps not, since we know it was inspired
      from QEDwhich was made multiple times by
      thompson and ritchiebefore ed was made.

```

15 | \*

Here we can see that we do the joining in two steps, first the lines 4 and 5 are joined, and then the newly modified line 4 and 5 are joined.

We can move a line from its current position to another line using the `m` command.

Lets insert a line-0 at the end of the file and then move it to the beginning of the file.

```

1 *7
2 line-4 end of file
3 *a
4 line-0 in the beginning, there was light
5 .
6 *8
7 line-0 in the beginning, there was light
8 *m0
9 *1,4p
10 line-0 in the beginning, there was light
11 line-1 hello universe
12 line-2 welcome to line editor
13 line-3 ed is perhaps the oldest editor out
      there
14 *
```

17: The undo command in ed is not as powerful as the undo command in vim. In vim, we can undo multiple changes using the `u` command. In ed, we can only undo the last change. If we run the `u` command multiple times, it will undo the last change of undoing the last change, basically redoing the last change.

We can also undo the last change using the `u` command.<sup>17</sup>

```

1 *1
2 line-0 in the beginning, there was light
3 *s/light/darkness
line-0 in the beginning, there was darkness
4 *u
5 *.
6 *.
7 line-0 in the beginning, there was light
8 *
```

If search and replace is not exactly what we want, and we want to totally change the line, we can use

the c command. It will let us type a new line, which will replace the current line.

```

1 *%p
2 line-0 in the beginning, there was light
3 line-1 hello universe
4 line-2 welcome to line editor
5 line-3 ed is perhaps the oldest editor out
   there
6 perhaps not, since we know it was inspired
   from QEDwhich was made multiple times by
   thompson and ritchiebefore ed was made.
7 line-4 end of file
8 *4
9 line-3 ed is perhaps the oldest editor out
   there
10 *C
11 line-4 ed is the standard editor for UNIX
12 .
13 *4
14 line-4 ed is the standard editor for UNIX
15 *

```

Just like the a command, we can also use the i command to insert a line at the current position. This will move the current line to the next line.

```

1 *6
2 line-4 end of file
3 *i
4 before end of file
5 .
6 *6,$p
7 before end of file
8 line-4 end of file
9 *

```

Finally, we can also number the lines using the n command.

```

1 *%p
2 line-0 in the beginning, there was light

```

```
3 line-1 hello universe
4 line-2 welcome to line editor
5 line-4 ed is the standard editor for UNIX
6 perhaps not, since we know it was inspired
   from QEDwhich was made multiple times by
   thompson and ritchiebefore ed was made.
7 before end of file
8 line-4 end of file
9 *%n
10 1      line-0 in the beginning, there was
    light
11 2      line-1 hello universe
12 3      line-2 welcome to line editor
13 4      line-4 ed is the standard editor for
    UNIX
14 5      perhaps not, since we know it was
    inspired from QEDwhich was made multiple
    times by thompson and ritchiebefore ed was
    made.
15 6      before end of file
16 7      line-4 end of file
17 *
```

### 2.2.3 Exploring Vim

There are a plethora of commands in vim. We wont be able to cover all of them in this course. Only the basic commands required to get started with using vim as your primary editor would be covered. A detailed tutorial on vim can be found by running the command `vimtutor` in your terminal.

```
1 | $ vimtutor
```

This opens a temporary files that goes through a lot of sections of vim, explaining the commands in detail. This opens the text file in vim itself, so you can actually try out each exercise as and when you read it. Many exercises are present in this file to help you remember and master commands. Feel free to modify the file since it is a temporary file and any changes made is lost if the command is re-run.

To open a file in vim, we provide the filename as an argument to the vim executable.

```
1 | $ vim test.txt
```

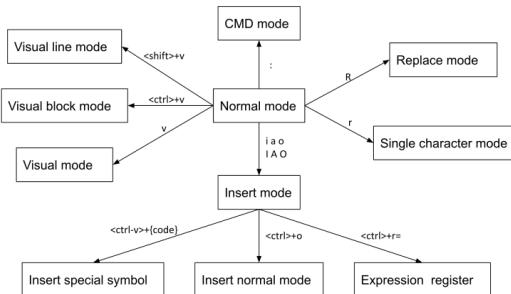
### Modal Editor

Vim is a modal editor, which means that it has different modes that it operates in. The primary modes are:

- ▶ Normal/Command Mode - The default mode where we can navigate around the file, and run vim commands.
- ▶ Insert Mode - The mode where we can type text into the file.
- ▶ Visual Mode - The mode where we can select text to copy, cut, or delete.
- ▶ Ex Mode - The mode where we can run ex commands.

Sometimes the normal mode is called command mode or escape mode, since we can run commands in this mode and we press the `Esc` key to go to this mode. However, the ex mode is also called command mode, since we can run ex commands in this mode. To avoid confusions, we will refer to the navigational(default) mode as normal mode, since vim internally also refers to it as normal mode, and we will refer to the ex mode as ex mode.

Pressing the Esc key takes you to the normal mode from any other mode.



**Figure 2.14:** Simplified Modes in Vim

18: This is a simplified version of the modes in vim. There are other interim modes and other keystrokes that toggle the modes. This is shown in detail in Figure 2.15.

The figure Figure 2.14 demonstrates how to switch between the different modes in vim.<sup>18</sup>

### Commands in Ex mode

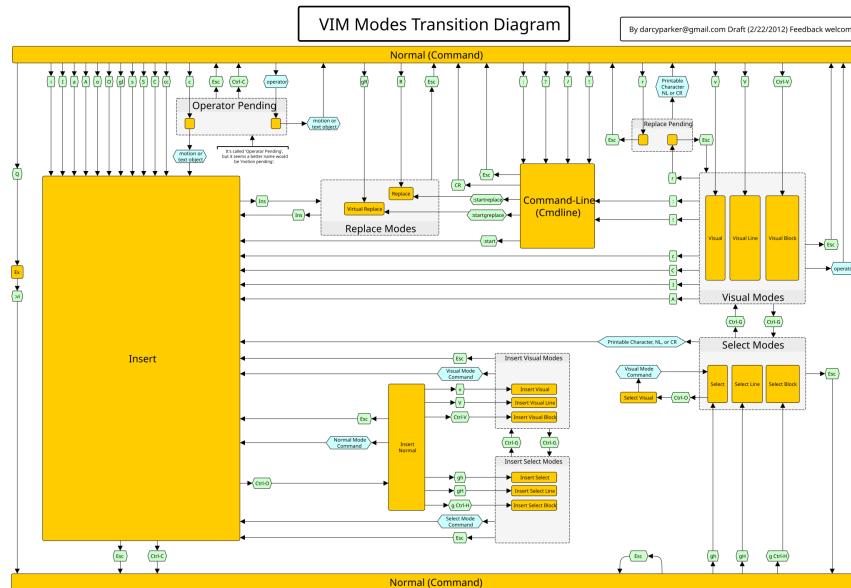
Since we are already familiar with commands in **ed**, most of the commands are same/similar in **ex** mode of vim.

There are many more commands in the **ex** mode of vim. Along with implementing the original **ed** commands, it also has a lot of additional commands to make it more integrated with the vim editor, such as the ability to open split windows, new tabs, buffers, and perform normal mode commands in **ex** mode.

### Basic Navigation

The basic keys for moving around in a text file in vim are the **h, j, k, l** keys. They move the cursor one character to the left, down, up, and right respectively. These keys are chosen because they are present on the home row of the keyboard, and do not require the user to move their hands from the home row to navigate.<sup>19</sup>

19: These were historically chosen because the ADM-3A terminal had these keys for navigation as seen in Figure 2.8.



**Figure 2.15:** Detailed Modes in Vim

Along with these, we have keys to navigate word by word, or to move to the next pattern match, or the next paragraph, spelling error, etc.

**Wait you forgot your cursor behind!**

All of the above commands move the cursor to the mentioned location. However, if you want to move the entire screen, and keep the cursor at its current position, you can use the z command along with t to top the line, b to bottom the line, and z to center the line.

There are other commands using the **Ctrl** key that moves the screen, and not the cursor.

## Replacing Text

Usually in other text editors, if you have a word, phrase, or line which you want to replace with

**Table 2.5:** Ex Commands in Vim

Key	Description
:f	show name of file
:p	print current line
:a	append at current line
:c	change current line
:d	delete current line
:i	insert line at current position
:j	join lines
:s	search and replace regex pattern in current line
:m	move current line to position
:u	undo latest change
:w [filename]	write buffer to filename
:q	quit if no change
:wq	write buffer to filename and quit
:x	write buffer to filename and quit
:q!	quit without saving
:r filename	read file contents into buffer
:r !command	read command output into buffer
:e filename	edit a file
:sp [filename]	split the screen and open another file
:vsp [filename]	vertical split the screen and open another file

another, you would either press the backspace or delete key to remove the text, and then type the new text. However, in vim, there is a more efficient way to replace text.

### Toggling Case

You can toggle the case of a character, word, line, or any arbitrary chunk of the file using the ~ or the g~ command.

You might start to see a pattern emerging here. Many commands in vim do a particular command, and on which text it operates is determined by the character followed by it. Such as c to change, d to delete, y to yank, etc. The text on which the

**Table 2.6:** Navigation Commands in Vim

Key	Description
h	move cursor left
j	move cursor down
k	move cursor up
l	move cursor right
w	move to the beginning of the next word
e	move to the end of the current word
b	move to the beginning of the previous word
\%	move to the matching parenthesis, bracket, or brace
0	move to the beginning of the current line
\$	move to the end of the current line
/	search forward for a pattern
?	search backward for a pattern
n	repeat the last search in the same direction
N	repeat the last search in the opposite direction
gg	move to the first line of the file
G	move to the last line of the file
1G	move to the first line of the file
1gg	move to the first line of the file
:1	move to the first line of the file
{	move to the beginning of the current paragraph
}	move to the end of the current paragraph
fg	move cursor to next occurrence of 'g' in the line
Fg	move cursor to previous occurrence of 'g' in the line

Key	Description
Ctrl+F	move forward one full screen
Ctrl+B	move backward one full screen
Ctrl+D	move forward half a screen
Ctrl+U	move backward half a screen
Ctrl+E	move screen up one line
Ctrl+Y	move screen down one line

**Table 2.7:** Moving the Screen Commands in Vim

command operates is mentioned using w for the word, 0 for till beginning of line, etc.

**Table 2.8:** Replacing Text Commands in Vim

Key	Description
r	replace the character under the cursor
R	replace the character from the cursor till escape is pressed
cw	change the word under the cursor
c4w	change the next 4 words
C	delete from cursor till end of line and enter insert mode
cc	delete entire line and enter insert mode
5cc	delete next 5 lines and enter insert mode
S	delete entire line and enter insert mode
s	delete character under cursor and enter insert mode

**Table 2.9:** Toggling Case Commands in Vim

Key	Description
~	toggle the case of the character under the cursor
g~w	toggle the case of the word under the cursor
g~0	toggle the case from cursor till beginning of line
g~\$	toggle the case from cursor till end of line
g~{	toggle the case from cursor till previous empty line
g~}	toggle the case from cursor till next empty line
g~\%	toggle the case from the bracket, brace, or parenthesis till its pair

This is not a coincidence, but rather a design of vim to make it more efficient to use. The first command is called the operator command, and the second command is called the motion command.

Vim follows a operator-count-motion pattern. For example: d2w deletes the next 2 words. This makes it very easy to learn and remember commands, since you are literally typing out what you want to do.

### Deleting or Cutting Text

In Vim, the delete command is used to cut text from the file.

### Motion - till, in, around

Key	Description
x	delete the character under the cursor
X	delete the character before the cursor
5x	delete the next 5 characters
dw	delete the word under the cursor
d4w	delete the next 4 words
D	delete from cursor till end of line
dd	delete entire line
6dd	delete next 6 lines

**Table 2.10:** Deleting Text Commands in Vim

By now you should notice that `dw` doesn't always delete the word under the cursor. Technically `dw` means delete till the beginning of the next word. So if you press `dw` at the beginning of a word, it will delete the word under the cursor. But if your cursor is in the middle of a word and you type `dw`, it will only delete the part of the word till the beginning of the next word from the cursor position.

To delete the entire word under the cursor, regardless of where the cursor is in the word, you can use the `diw` this means **delete inside word**.

However, now you may notice that `diw` doesn't delete the space after the word. This results in two consecutive spaces, one from the end of the word, and one from the beginning of the word being left behind. To delete the space as well, you can use `daw` which means **delete around word**.

This works not just with `w` but with any other motion such as **delete inside paragraph**, which will delete the entire paragraph under the cursor, resulting in two empty lines being left behind, and **delete around paragraph**, which will delete the entire paragraph under the cursor, and only one empty line being left behind.

Try out the same with deleting inside other items,

such as brackets, parenthesis, braces, quotes, etc. The syntax remains the same, `di{`, `di[`, `di(`, `di"`, `di'`, etc.

### Yanking and Pasting Text

Yes, copying is called yanking in vim. The command to yank is `y` and to paste is `p`. You can combine `y` with all the motions and **in** and **around** motions as earlier. You can also add the count to yank multiple lines or words.

**Table 2.11:** Deleting Text Commands in Vim

Key	Description
<code>yy</code>	yank the entire line
<code>yw</code>	yank the word under the cursor
<code>:</code>	<code>:</code>
<code>p</code>	paste the yanked text after the cursor
<code>P</code>	paste the yanked text before the cursor

**Remark 2.2.2** Important to note that the commands

1 | `yy`

and

1 | `0y$`

are not the same. The first command yanks the entire line, including the newline character at the end of the line. The second one yanks the entire line, but does not include the newline character at the end of the line. Thus if you directly press `p` after the first command, it will paste the line below the current line, and if you press `p` after the second command, it will paste the line at the end of the current line.

### Undo and Redo

The undo command in vim is `u` and the redo com-

mand is `Ctrl+R`. You can undo multiple changes, unlike `ed`.

**Remark 2.2.3** If you want to use vim as your primary editor, it is highly recommended to install the `mbbill/undotree` plugin. This plugin will show you a tree of all the changes you have made in the current buffer, and you can go to any point in the tree and undo or redo changes. This becomes very useful if you undo too many changes and by mistake make a new change, this changes your branch in undo tree, and you cannot redo the changes you undid. With the `undotree` plugin, you can switch branches of the undo tree and redo the changes.

## Searching and Replacing

The search command in vim is `/` for forward search and `?` for backward search. You can use the `n` command to repeat the last search in the same direction, and the `N` command to repeat the last search in the opposite direction. For example, if you perform forward search then using the `n` command will search for the next occurrence of the pattern in the forward direction, and using the `N` command will search for the previous occurrence of the pattern in the backward direction. However if you perform a backward search using the `?` command, then using the `n` command will search for the previous occurrence of the pattern in the backward direction, and using the `N` command will search for the next occurrence of the pattern in the forward direction.

You can also use the `*` command to search for the word under the cursor, and the `#` command to search for the previous occurrence of the word under the cursor.

You can perform search and replace using the `:s` command. The command takes a line address on which to perform the search and replace. Usually you can use the `\%` address to search in the entire file, or the `.,$` address to search from cursor till the end of the file.

You can also use any line number to specify the address range, similar to the ed editor.

`1 | :[addr]s/pattern/replace/[flags]`

The flags at the end of the search and replace command can be `g` to replace all occurrences in the line, and `c` to confirm each replacement.

The address can be a single line number, a range of line numbers, or a pattern to search for. The pattern can be a simple string, or a regular expression.

Some examples of addresses are shown in Table 2.12.

**Table 2.12:** Address Types in Search and Replace

Key	Description
<code>m,n</code>	from line m to line n
<code>m</code>	line m
<code>m,\$</code>	from line m to end of file
<code>.,\$</code>	from current line to end of file
<code>1,n</code>	from line 1 to line n
<code>/regex/,n</code>	from line containing regex to line n
<code>m,/regex/</code>	from line m to line containing regex
<code>./regex/</code>	from current line to line containing regex
<code>/regex//.</code>	from line containing regex to current line
<code>1,/regex/</code>	from the first line to line containing regex
<code>/regex/,\$</code>	from line containing regex to the last line
<code>/regex1/;/regex2/</code>	from line containing regex1 to line containing regex2
<code>\%</code>	entire file

## Insert Mode

**Table 2.13:** Keys to enter Insert Mode

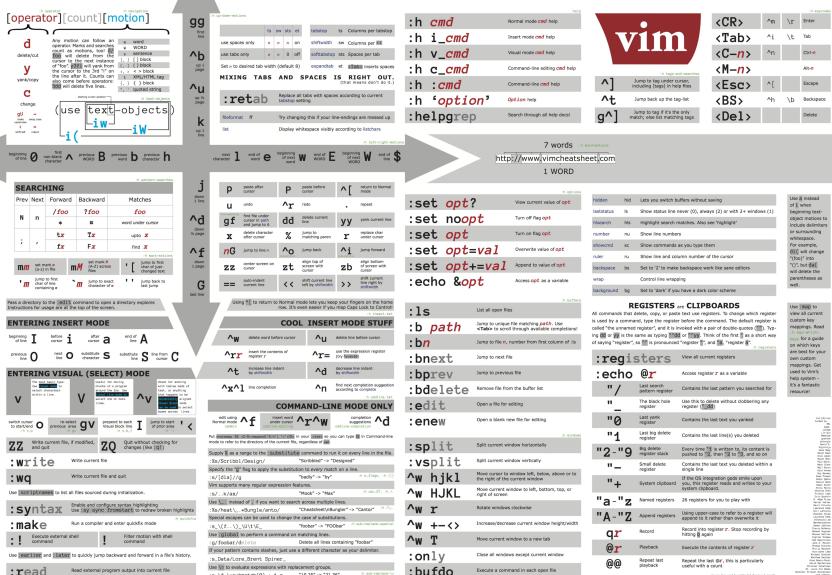
Key	Description
i	enter insert mode before the cursor
a	enter insert mode after the cursor
I	enter insert mode at the beginning of the line
A	enter insert mode at the end of the line
o	add new line below the current line and enter insert mode
O	add new line above the current line and enter insert mode

You can enter insert mode from escape mode using the keys listed in Table 2.13. In insert mode, if you want to insert any non-graphical character, you can do that by pressing `Ctrl+V` followed by the key combination for the character. For example, to insert a newline character, you can press `Ctrl+V` followed by `Enter`.

These are just the basic commands to get you started with vim. You can refer to vim cheat sheets present online to get more familiar with the commands.

- ▶ <https://vim.rtorr.com/> is a good text based HTML cheat sheet for vim.
- ▶ <https://vimcheatsheet.com/> is a paid graphical cheat sheet for vim.<sup>20</sup>

<sup>20</sup>: A free version of the graphical cheat sheet is shown in Figure 2.16.



**Figure 2.16:** Vim Cheat Sheet

## 2.3 Emacs

### 2.3.1 History

Emacs was mostly developed by Richard Stallman and Guy Steele.

**Table 2.14:** History of Emacs

<b>1962</b> .....	TECO (Tape Editor and Corrector) was developed at MIT.
<b>1976</b> .....	Richard Stallman visits Stanford AI Lab and sees Fred Wright's E editor.
<b>1978</b> .....	Guy Steele accumulates a collection of TECO macros into EMACS.
<b>1979</b> .....	EMACS becomes MIT's standard text editor.
<b>1981</b> .....	James Gosling writes Gosling Emacs that runs on UNIX.
<b>1984</b> .....	Richard Stallman starts GNU Emacs - a free software alternative to Gosling Emacs.

#### TECO

TECO was developed at MIT in 1962. It was a text editor used to correct the output of the PDP-1 computer. It is short for Tape Editor and Corrector. Unlike most modern text editors, TECO used separate modes in which the user would either add text, edit existing text, or display the document. One could not place characters directly into a document.



**Figure 2.17:** Richard Stallman - founder of GNU and FSF projects

by typing them into TECO, but would instead enter a character ('i') in the TECO command language telling it to switch to input mode, enter the required characters, during which time the edited text was not displayed on the screen, and finally enter a character (<esc>) to switch the editor back to command mode. This is very similar to how vi works.

### **Stallman's Visit to Stanford**

21: What You See Is What You Get

In 1976, Richard Stallman visited the Stanford AI Lab where he saw Fred Wright's E editor. He was impressed by E's WYSIWYG<sup>21</sup> interface where you do not need to tackle multiple modes to edit a text file. This is the default behaviour of most modern editors now. He then returned to MIT where he found that Carl Mikkelsen had added to TECO a combined display/editing mode called Control-R that allowed the screen display to be updated each time the user entered a keystroke. Stallman reimplemented this mode to run efficiently and added a macro feature to the TECO display-editing mode that allowed the user to redefine any keystroke to run a TECO program.

Initially TECO was able to only edit the file sequentially, page by page. This was due to earlier memory restrictions of the PDP-1. Stallman modified TECO to read the entire file into the buffer, and then edit the buffer in memory allowing for random access to the file.

### **Too Many Macros!**

The new version of TECO quickly became popular at the AI Lab and soon accumulated a large collection of custom macros whose names often ended in MAC or MACS, which stood for macro. This quickly got out of hand as there were many divergent macros,

and a user would be totally lost when using a co-worker's terminal.

In 1979, Guy Steele combined many of the popular macros into a single file, which he called EMACS, which stood for Editing MACroS, or E with MACroS.

To prevent thousands of forks of EMACS, Stallman declared that 'EMACS was distributed on a basis of communal sharing, which means all improvements must be given back to me to be incorporated and distributed.'

Till now, the EMACS, like TECO, ran on the PDP-10 which ran the ITS operating system and not UNIX.

### EINE ZWEI SINE and other clones

No, that is not German. These are some of the popular clones of EMACS made for other operating systems.

EINE<sup>22</sup> was a text editor developed in the late 1970s. In terms of features, its goal was to 'do what Stallman's PDP-10 (original) Emacs does'. Unlike the original TECO-based Emacs, but like Multics Emacs, EINE was written in Lisp. It used Lisp Machine Lisp.

In the 1980s, EINE was developed into ZWEI<sup>23</sup>. Innovations included programmability in Lisp Machine Lisp, and a new and more flexible doubly linked list method of internally representing buffers.

SINE<sup>24</sup> was written by Owen Theodore Anderson in 1981.



**Figure 2.18:** Guy L. Steele Jr. combined many divergent TECO with macros to create EMACS

22: EINE stands for Eine Is Not EMACS

23: ZWEI stands for ZWEI Was Eine Initially

These kinds of recursive acronyms are common in the nix world. For example, GNU stands for GNU's Not Unix, WINE (A compatibility layer to run Windows applications) is short for WINE Is Not an Emulator.

24: SINE stands for SINE Is Not EINE

In 1978, Bernard Greenberg wrote a version of EMACS for the Multics operating system called Multics EMACS. This used Multics Lisp.



**Figure 2.19:** James Gosling - creator of Gosling Emacs and later Java

25: Recall from the previous chapter that free software does not mean a software provided gratis, but a software which respects the user's freedom to run, copy, distribute, and modify the software. It is like free speech, not free beer.

### Gosling Emacs

In 1981, James Gosling wrote Gosling Emacs for UNIX. It was written in C and used Mocklisp, a language with lisp-like syntax, but not a lisp. It was not free software.

### GNU Emacs

In 1983, Stallman started the GNU project to create a free software alternatives to proprietary softwares and ultimately to create a free<sup>25</sup> operating system.

In 1984, Stallman started GNU Emacs, a free software alternative to Gosling Emacs. It was written in C and used a true Lisp dialect, Emacs Lisp as the extension language. Emacs Lisp was also implemented in C. This is the version of Emacs that is most popular today and available on most operating systems repositories.

### How the developer's keyboard influences the editors they make

Remember that ed was made while using **ADM-3A** which looked like Figure 2.20.

Whereas emacs was made while the **Knight keyboard** and the **Space Cadet keyboard** were in use, which can be seen in Figure 2.21.

Notice how the ADM-3A has very limited modifier keys, and does not even have arrow keys. Instead it uses h, j, k, l keys as arrow keys with a modifier. This is why vi uses mostly key combinations and a modal interface. Vi also uses the Esc key to switch between modes, which is present conveniently in



**Figure 2.20:** ADM-3A terminal



**Figure 2.21:** Space Cadet Keyboard

place of the Caps Lock or Tab key in modern keyboard layouts.

The Space Cadet keyboard has many modifier keys, and even a key for the Meta key. This is why emacs uses many key modifier combinations, and has a lot of keybindings.

### 2.3.2 Exploring Emacs

This is not a complete overview of Emacs, or even its keybindings. A more detailed reference card can be found on their [website](#).

#### Opening a File

We can open a file in emacs by providing its filename as an argument to the emacs executable.

```
| $ emacs test.txt
```

Most of emacs keybindings use modifier keys such as the `Ctrl` key, and the `Meta` key. The `Meta` key is usually the `Alt` key in modern keyboards. In the reference manual and here, we will be representing the `Meta` key as `M-` and the `Ctrl` key as `C-`.

#### Basic Navigation

These keys are used to move around in the file. Like vim, emacs also focusses on keeping the hands free from the mouse, and on the keyboard. All the navigation can be done through the keyboard.

#### Exiting Emacs

We can exit emacs either with or without saving the file. We can also suspend emacs and return to the shell. This is a keymapping of the shell, and not of emacs.

#### Searching Text

**Table 2.15:** Navigation Commands in Emacs

Key	Description
C-p	move up one line
C-b	move left one char
C-f	move right one char
C-n	move down one line
C-a	goto beginning of current line
C-e	goto end of current line
C-v	move forward one screen
M-<	move to first line of the file
M-b	move left to previous word
M-f	move right to next word
M->	move to last line of the file
M-a	move to beginning of current sentence
M-e	move to end of current sentence
M-v	move back one screen

**Table 2.16:** Exiting Emacs Commands

Key	Description
C-x C-s	save buffer to file
C-z	suspend emacs
C-x C-c	exit emacs and stop it

Emacs can search for a fixed string, or a regular expression and replace it with another string.

**Table 2.17:** Searching Text Commands in Emacs

Key	Description
C-s	search forward
C-r	search backward
M-x	replace string

### Copying and Pasting

Copying can done by marking the region, and then copying it.

Key	Description
M-backspace	cut the word before cursor
M-d	cut the word after cursor
M-w	copy the region
C-w	cut the region
C-y	paste the region
C-k	cut from cursor to end of line
M-k	cut from cursor to end of sentence

**Table 2.18:** Copying and Pasting Commands in Emacs

## 2.4 Nano

Although vim and emacs are the most popular command line text editors, nano is also a very useful text editor for beginners. It is very simple and does not have a steep learning curve.

It is a non-modal text editor, which means that it does not have different modes for different actions. You can directly start typing text as soon as you open **nano**.

Although it uses modifier keys to invoke commands, it does not have a lot of commands as vim or emacs.

### 2.4.1 History

Pine<sup>26</sup> was a text-based email client developed at the University of Washington. It was created in 1989. The email client also had a text editor built in called Pico.

Although the license of Pine and Pico may seem open source, it was not. The license was restrictive and did not allow for modification or redistribution.

<sup>27</sup>



**Figure 2.22:** Nano Text Editor

<sup>26</sup>: It is believed that pine stands for Pine is Not Elm, Elm being another text-based email client. However, the author clarifies that it was not named with that in mind. Although if a backronym was to be made, he preferred 'Pine is Nearly Elm' or 'Pine is No-longer Elm'

27: Up to version 3.91, the Pine license was similar to BSD, and it stated that 'Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee to the University of Washington is hereby granted ...' The university registered a trademark for the Pine name with respect to 'computer programs used in communication and electronic mail applications' in March 1995. From version 3.92, the holder of the copyright, the University of Washington, changed the license so that even if the source code was still available, they did not allow modifications and changes to Pine to be distributed by anyone other than themselves. They also claimed that even the old license never allowed distribution of modified versions.

28: Mathematically, nano is  $10^{-9}$  or one billionth. and pico is  $10^{-12}$  or one trillionth. or put relatively, nano is 1000 times bigger than pico, although the size of nano binary is smaller than pico.

Due to this, many people created clones of Pico with free software licenses. One of the most popular clones was TIP (TIP isn't Pico) which was created by Chris Allegretta in 1999. Later in 2000 the name was changed to Nano.<sup>28</sup> In 2001, nano became part of the GNU project.

GNU nano implements several features that Pico lacks, including syntax highlighting, line numbers, regular expression search and replace, line-by-line scrolling, multiple buffers, indenting groups of lines, rebindable key support, and the undoing and redoing of edit changes.

In most modern linux systems, the **nano** binary is present along with the **pico** binary, which is actually a symbolic link to the **nano** binary.

You can explore this by finding the path of the executable using the `which` command and long-listing the executable.

```

1 $ which pico
2 /usr/bin/pico
$ ls -l /usr/bin/pico
lrwxrwxrwx 1 root root 22 Sep  6  2023 /usr/
      bin/pico -> /etc/alternatives/pico
$ ls -l /etc/alternatives/pico
lrwxrwxrwx 1 root root 9 Sep  6  2023 /etc/
      alternatives/pico -> /bin/nano

```

**Remark 2.4.1** Note that here we have a symlink to another symlink. Theoretically, you can extend to as many levels of chained symlinks as you want. Thus, to find the final sink of the symlink chain, you can use the `readlink -f` command or the `realpath` command.

```

1 $ realpath $(which pico)
2 /usr/bin/nano

```

## 2.4.2 Exploring Nano

In nano, the Control key is represented by the ^symbol. The Meta or Alt key is represented by the M-.

### File Handling

You can open a file in nano by providing the file-name as an argument to the nano executable.

```
1 | $ nano test.txt
```

Key	Description
^S	save the file
^O	save the file with a new name
^X	exit nano

**Table 2.19:** File Handling Commands in Nano

### Editing

Nano is a simple editor, and you can do without learning any more commands than the ones listed above, but here are some more basic commands for editing text.

Key	Description
^K	cut current line and save in cutbuffer
M-6	copy current line and save in cutbuffer
^U	paste contents of cutbuffer
M-T	cut until end of buffer
^]	complete current word
M-U	undo last action
M-E	redo last undone action
^J	justify the current paragraph
M-J	justify the entire file
M-:	start/stop recording a macro
M-;	run the last recorded macro
F12	invoke the spell checker, if available

**Table 2.20:** Editing Commands in Nano

You can also find third-party cheat sheets [online](#).

There are many more commands in nano, but they are omitted from here for brevity. You can find the complete list of keybindings by pressing  $\text{^G}$  key in nano, or by running `info nano`.

### 2.4.3 Editing A Script in Nano

Since learning nano is mostly to be able to edit a text file even if you are not familiar with either vim or emacs, let us try to edit a simple script file to confirm that you can use nano.

```
1 $ touch myscript.sh
2 $ chmod u+x myscript.sh
3 $ nano myscript.sh
```

Now try to write a simple script in the file. An example script is shown below.

```
1 #!/bin/bash
2 read -rp 'What is your name? ' name
3 echo "Hello $name"
4 date=$(date "+%H:%M on a %A")
5 echo "Currently it is $date"
```

If you do not understand how the script works, do not worry. It will be covered in depth in later chapters.

Now save the file by pressing  $\text{^S}$

**Remark 2.4.2** In some systems, the  $\text{^S}$  key will freeze the terminal. Any key you press after this will seem to not have any effect. This is because it is interpreted as the  $XOFF$  and is used to lock the scrolling of the terminal. To unfreeze the terminal, press  $\text{^Q}$ . In such a system, you can save the file by pressing  $\text{^O}$  and then typing out the name of the file if not present already, and pressing Enter. To disable this behaviour, you can add the line

```
1 | set -ixon
```

to your .bashrc file.

and exit nano by pressing ^X.

Now you can run the script by typing

```
1 $ ./myscript.sh
2 What is your name? Sayan
3 Hello Sayan
4 Currently it is 21:50 on a Tuesday
```

Now that we are able to edit a text file using text editors, we are ready to write scripts to solve problems.



# Networking and SSH

## 3.1 Networking

### 3.1.1 What is networking?

Have you ever tried to get some work done on a computer while the internet was down? It's a nightmare. Modern day computing relies highly on networking. But what is networking?

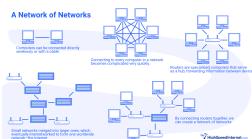
**Definition 3.1.1** (Networking) A computer network comprises two or more computers that are connected—either by cables (wired) or wifi (wireless)—with the purpose of transmitting, exchanging, or sharing data and resources.

We have been using the computer, and linux, for a while now but the utility of a computer increases exponentially when it is connected to a network. It allows computers to share files and resources, and to communicate with each other. Current day world wide web is built on the internet.

**Definition 3.1.2** (Internet) Internet is a global network of networks that connects millions of computers worldwide. It allows computers to connect to other computers across the world through a hierarchy of routers and servers.

Learning about networking and how networking works is useful, although we won't be devling into details in this book. It is left as an exercise for the

One succinct blogpost explaining how the internet works from which the figure Figure 3.1 is taken is available at <https://www.highspeedinternet.com/resources/how-the-internet-works>



**Figure 3.1:** Types of Networks

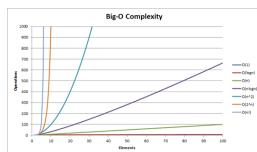
reader to explore external resources if they are interested.

### 3.1.2 Types of Networks

If the end goal is to connect computers with each other, one naive solution might be to connect all the computers with each other. Although this might seem intuitive at first, this quickly gets out of hand when the number of computers keep increasing.

If we have  $n$  computers, then the number of connections required to connect all the computers with each other is given by the formula

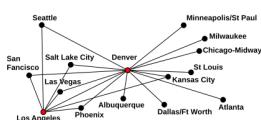
$$\frac{n(n - 1)}{2} = \frac{n^2 - n}{2}$$



**Figure 3.2:** Growth Rate of Different Functions - Note how quickly  $n^2$  grows

This is a quadratic function and grows very quickly.

This means it will cost a lot to connect all the computers to each other. This is applicable not only in computer networking with physical wires, but in many other fields. Take an examples of airlines and airplane routes. If there were  $n$  airports, then the number of routes required to connect all the airports is given by the same formula. This would be disastrous for the economy and the environment if we ran so many airplanes daily. So what gives?



**Figure 3.3:** Hub and Spoke Model Employed by Airlines

### Hub and Spoke Network

The solution to this problem is to use a hub and spoke model, where there are one, or multiple, central hubs which connect to many other nodes. Any path from any node to another goes through one or more hubs. This reduces the number of connections required to connect all the nodes.

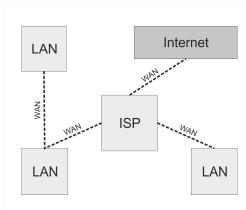
This is the solution used in airlines, and also in most computer networks.<sup>1</sup>

Due to this, networks can be classified into three broad categories based on their geographical area coverage.

- ▶ **Local Area Network (LAN):** A network that covers a small geographical area, like a home, office, or a building.
- ▶ **Metropolitan Area Network (MAN):** A network that covers a larger geographical area, like a city or a town.
- ▶ **Wide Area Network (WAN):** A network that covers a large geographical area, like a country or the entire world.

To connect these networks to computers and also to each other, we require some special devices.

1: Although computer networks use a hub model for the local area network, the network of networks, especially the gateway routers follow a mesh model to ensure redundancy and make the network more robust.



**Figure 3.4:** LAN and WAN connecting to the Internet

### 3.1.3 Devices in a Network

In computer networks, this hub of the Hub and Spoke Model can either be a level 1 hub, a level 2 switch, or a level 3 router.

#### Hub

A hub will simply broadcast the message to all the connected nodes. This causes a lot of traffic to be generated and is not very efficient. Hub does not have the capability to identify which node is who. This is called a level 1 hub.<sup>2</sup>

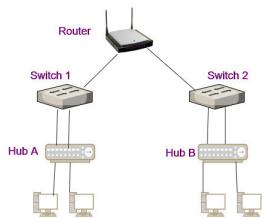
2: To understand more about the levels, refer [OSI Model](#)

#### Switch

A switch is smarter than a hub. It can identify each device connected to it and can send the packets of data only to the intended recipient. This is more efficient than a hub. This is called a level 2 switch

since it uses the level 2 of the OSI model (Data Link Layer) to identify the devices. This means that the devices are identified by their MAC addresses. Using this, you can only communicate with devices in your local network. This is useful for a home network or a office network. But we cannot communicate with the entire world using this, since it doesn't understand IP addresses.

## Router



**Figure 3.5:** Hub, Switch, Router connecting to the Internet

A router is even smarter than a switch. It can understand IP addresses and can route the packets from one network to another. This is called a level 3 router since it uses the level 3 of the OSI model (Network Layer) to identify the devices. This means that the networks are identified by their IP addresses. This is what we use to connect to the internet. The internet is nothing but a whole lot of routers communicating with each other to find the optimal path to send the packets to its destination. Border Gateway Protocol (BGP) is the protocol used by routers to communicate with each other and find the optimal path. They usually are connected in a mesh network to ensure redundancy and robustness.

## Level 3 Switch

A level 3 switch, or a routing switch, is a switch with the capabilities of a router. It can understand the language of IP addresses and can route the packets to different networks. This is useful in large organizations where there are many networks and each network can be divided into subnetworks called VLANs.

You can read more about the differences between these devices [online](#).

So, in short, internet is a network of network of . . . networks. It is a hierarchical structure connecting all the computers in the world. Some computers are connected earlier in the hierarchy (usually the

ones closer geographically) and some are connected later.

### 3.1.4 IP Addresses

So how do routers know which where to send the data packets? This is where IP addresses come in. To communicate over the internet, two computers need to know their public IP addresses. The routers then finds the optimal path to send the data packets to the destination network.

IP addresses are of two types: IPv4 and IPv6. The most common one is IPv4, which is a 32-bit address represented as four octets separated by dots. For example,

162.136.73.21

Here, each octet can take values from 0 to 255.<sup>3</sup> Technically all such combinations are possible IP addresses, resulting in  $2^{32} = 4,294,967,296$  possible IP addresses. That is a lot of IP addresses, but not enough for the growing number of devices in the world.

This is where IPv6 comes in. IPv6 is a 128-bit address represented as 8 groups of 4 hexadecimal digits separated by colons. For example,

2001 : 0db8 : 85a3 : 0000 : 0000 : 8a2e : 0370 : 7334

This results in

$$2^{128} = 340282366920938463463374607431768211456$$

<sup>3</sup>: An octet is a group of 8 bits. Since an IP address is 32 bits, it is represented as 4 groups of 8 bits. 8 bits can only represent numbers from 0 to 255. since  $2^8 = 256$ . Notice that there are some groups of zeros in the address. These can be compressed by writing only one zero in place of multiple zeros in each group. Further, any leading zeros can be omitted for each group. Making the above address as 2001:db8:85a3:0:0:8a2e:370:7334. Further, if there are multiple groups of zeros, they can be compressed to ::. This can be used only once in an address. Doing this, the above address can be compressed further to 2001:db8:85a3::8a2e:370:7334.

possible IP addresses, which is a lot more than IPv4.

### 3.1.5 Subnetting

#### Legacy Classes

10.125.42.62 → 00001010.01111101.00101010.00111110

Recall that an IP address, although represented as four octets, is actually a 32-bit address. This means that in binary form, an IP address is a string of 32 1s and 0s. Using the first four bits, we can classify an IP address into five classes.

- ▶ **Class A:** The first bit is ‘0’. The IP addresses in the range 0.0.0.0 to 127.255.255.255.
- ▶ **Class B:** The first two bits are ‘10’. IP addresses in the range 128.0.0.0 to 191.255.255.255.
- ▶ **Class C:** The first three bits are ‘110’. IP addresses in the range 192.0.0.0 to 223.255.255.255.
- ▶ **Class D:** The first four bits are ‘1110’. IP addresses in the range 224.0.0.0 to 239.255.255.255. These are reserved for multicast addresses.
- ▶ **Class E:** The first four bits are ‘1111’. IP addresses in the range 240.0.0.0 to 255.255.255.255. These are reserved for experimental purposes.

However, these classes do not simply assign an IP to each machine. They are further divided into the network part and the host part.

Class A assigns the first octet to the network part, this is used to identify which network the machine is in. The remaining three octets are used to identify the host in that network. This means that a class A

network can have  $2^{24} - 2 = 16,777,214$  hosts. However, there can only be  $2^7 = 128$  class A networks. Thus, class A networks are used by large organizations which have many hosts, but not many large organizations exist, so 128 networks are enough.

Similarly, class B assigns the first two octets to identify the network and the remaining two octets to identify the host. This means that a class B network can have  $2^{16} - 2 = 65,534$  hosts. And there can be  $2^{14} = 16,384$  class B networks. These are used by medium-sized organizations, which are plenty in number, and have a moderate number of hosts.

The same goes for class C networks, where the first three octets are used to identify the network and the last octet is used to identify the host. This network can have  $2^8 - 2 = 254$  hosts. And there can be  $2^{21} = 2,097,152$  class C networks. These are used by small organizations, which are plenty in number, and have a small number of hosts.

### Subnet Masks

**Definition 3.1.3** (Subnetting) The process of dividing a network into smaller network sections is called subnetting.

Usually, each network has only one subnet, which contains all the hosts in that network. However, the network can be further divided into smaller subnetworks, each containing a subset of the hosts. This is useful in large organizations where the network is divided into departments, and each department is given a subnetwork.

To indicate which part of the IP address is the network part and which part is the host part, we use a subnet mask. A subnet mask is a 32-bit number

where the first  $n$  bits are 1s and the remaining bits are 0s. The number of 1s in the subnet mask indicates the number of bits used to identify the network. For example, for the IP Address 192.168.0.15, it can be written in binary as

$$11000000 - 10101000 - 00000000 - 00001111$$

As we know, it belongs to the class C network, where the first three octets are used to identify the network, and the rest is used to identify the host. So the default network mask is

$$11111111 - 11111111 - 11111111 - 00000000$$

or 255.255.255.0 in decimal. The network portion of the IP address is found by taking the bitwise AND<sup>4</sup> of the IP address and the subnet mask. This results in the network address

$$11000000 - 10101000 - 00000000 - 00000000$$

which is 192.168.0.0 and the host address is 0000 1111 which is 15.

However, if we do not require all the 8 bits in the host space<sup>5</sup> then we can use some of the initial bits of the host space to identify subnetworks. This is called subnetting.

For example, the netmask of 255.255.255.0 leaves 8 bits for the host space, or  $2^8 - 2 = 254$ <sup>6</sup> hosts. If we want to split this network into two subnets, we can use the MSB of the host space for representing the subnetworks. This results in each subnet having  $2^7 - 2 = 126$  hosts.

- 4: The bitwise AND operation is a binary operation that takes two equal-length binary representations and performs the logical AND operation on each pair of corresponding bits. The result in each position is 1 if the first bit is 1 and the second bit is 1; otherwise, the result is 0.
- 5: That is, if we have less than 254 hosts in the network.
- 6: We subtract 2 from the total number of hosts to account for the network address and the broadcast address, which are the first(0) and the last(255) addresses in the network.

**Remark 3.1.1** Observe that we effectively lost two available addresses from the total number of hosts in the network. Earlier we could have 254 hosts, but now we can have only  $126 \times 2 = 252$  hosts. This is because each subnet also reserves the first and the last address for the network address and the broadcast address.

To do this, we change the subnet mask to

11111111 – 11111111 – 11111111 – 10000000

which can be represented as 255.255.255.128.

This gives us two subnets, one with the address range of 192.168.0.1 to 192.168.0.127, and another of 192.168.0.129 to 192.168.0.255.

### 3.1.6 Private and Public IP Addresses

But what if we want to communicate with computers in our local network? This is where private IP comes in. Some ranges of IP addresses are reserved for private networks. These are not routable over the internet. Each LAN has a private IP address range, and the router translates these private addresses to the public IP address when sending the packets over the internet. The assignment of these private IP addresses is done by the DHCP server<sup>7</sup> in the router.

Each class of networks has a range of IP addresses that are reserved for private networks.

7: Dynamic Host Configuration Protocol (DHCP) is a network management protocol used on Internet Protocol networks whereby a DHCP server dynamically assigns an IP address and other network configuration parameters to each device on a network so they can communicate with other IP networks.

**Table 3.1:** Private IP Address Ranges

Class	Network Bits	Address Range	Number of Addresses
Class A	8	10.0.0.0 - 10.255.255.255	16,777,216
Class B	12	172.16.0.0 - 172.31.255.255	1,048,576
Class C	16	192.168.0.0 - 192.168.255.255	65,536

### 3.1.7 CIDR

However, this practice of subdividing IP addresses into classes is a legacy concept, and not followed anymore. Instead, we use CIDR (Classless Inter-Domain Routing) to announce how many bits are used to identify the network and how many bits are used to identify the host.

For example, we could express the idea that the IP address 192.168.0.15 is associated with the netmask 255.255.255.0 by using the CIDR notation of 192.168.0.15/24. This means that the first 24 bits of the IP address given are considered significant for the network routing.

This is helpful because not all organizations fit into the tight categorization of the legacy classes.

### 3.1.8 Ports

Ports usually refer to physical holes in a computer where you can connect a cable. However, in networking, ports refer to logical endpoints for communication.

**Definition 3.1.4 (Port)** A port or port number is a number assigned to uniquely identify a connection endpoint and to direct data to a

specific service. At the software level, within an operating system, a port is a logical construct that identifies a specific process or a type of network service.

For any communication between two computers, the data needs to be sent to a specific port. This is because there are multiple services running on a computer, and the operating system needs to know which service to direct the data to.

There are  $2^{16} = 65,536$  ports available for use. However, the first 1024 ports are reserved for well-known services. These are called the **well-known ports** and are used by services like HTTP, FTP, SSH, etc. The well known ports can be found in Table 3.2.

Other ports, from 1024 to 49151, are registered ports, these ports can be registered with the Internet Assigned Numbers Authority (IANA) by anyone who wants to use them for a specific service.

Ports from 49152 to 65535 are dynamic ports, these are used by the operating system for temporary connections and are not registered.

Whenever you send a request to a web server for example, the request is sent to the server's IP address and the port number 80 which is the default port for HTTP<sup>8</sup> but the port number from your (the client) side is usually a random port number from the dynamic port range. This is a short-lived port number and is used to establish a connection with the server.

8: or 443 for HTTPS.

These kinds of ports which are short-lived and used to establish a connection are called **ephemeral ports**.

**Table 3.2:** Well-known Ports

Port Number	Service	Protocol
20	File Transfer Protocol (FTP) Data Transfer	TCP
21	File Transfer Protocol (FTP) Command Control	TCP
22	Secure Shell (SSH) Secure Login	TCP
23	Telnet remote login service	TCP
25	Simple Mail Transfer Protocol (SMTP)	TCP
53	Domain Name System (DNS) service	TCP/UDP
67, 68	Dynamic Host Configuration Protocol (DHCP)	UDP
80	Hypertext Transfer Protocol (HTTP)	TCP
110	Post Office Protocol (POP3)	TCP
119	Network News Transfer Protocol (NNTP)	TCP
123	Network Time Protocol (NTP)	UDP
143	Internet Message Access Protocol (IMAP)	TCP
161	Simple Network Management Protocol (SNMP)	UDP
194	Internet Relay Chat (IRC)	TCP
443	HTTP Secure (HTTPS) HTTP over TLS/SSL	TCP
546, 547	DHCPv6 IPv6 version of DHCP	UDP

### 3.1.9 Protocols

**Definition 3.1.5** (Protocols) In computing, a protocol is a set of rules that define how data is transmitted between devices in a network.

There are many protocols used in networking, some of the most common ones are

- ▶ **HTTP:** HyperText Transfer Protocol
- ▶ **HTTPS:** HyperText Transfer Protocol Secure
- ▶ **FTP:** File Transfer Protocol
- ▶ **SSH:** Secure Shell
- ▶ **SMTP:** Simple Mail Transfer Protocol
- ▶ **POP3:** Post Office Protocol
- ▶ **IMAP:** Internet Message Access Protocol
- ▶ **DNS:** Domain Name System
- ▶ **DHCP:** Dynamic Host Configuration Protocol

- ▶ **NTP:** Network Time Protocol
- ▶ **SNMP:** Simple Network Management Protocol
- ▶ **IRC:** Internet Relay Chat
- ▶ **BGP:** Border Gateway Protocol
- ▶ **TCP:** Transmission Control Protocol
- ▶ **UDP:** User Datagram Protocol

These protocols act on the different layers of the OSI model. For example, HTTP, HTTPS, FTP, SSH, etc. are application layer protocols, while TCP, UDP, etc. are transport layer protocols.

### 3.1.10 Firewalls

**Definition 3.1.6 (Firewall)** A firewall is a network security system that monitors and controls incoming and outgoing network traffic based on predetermined security rules.

A firewall acts as a barrier between your computer and the internet. It monitors the incoming and outgoing traffic and blocks any traffic that does not meet the security rules. This is useful to prevent unauthorized access to your computer and to prevent malware from entering your computer and/or communicating with the outside world.

```
1 $ sudo ufw enable # Enable the firewall
2 $ sudo ufw allow 22 # Allow SSH
3 $ sudo ufw allow 80 # Allow HTTP
4 $ sudo ufw allow 443 # Allow HTTPS
5 $ sudo ufw status # Check the status of the
firewall
```

### 3.1.11 SELinux

We can have additional security by using SELinux in addition to the firewall. SELinux is a security module that provides access control security policies. SELinux is short for Security-Enhanced Linux. It provides a flexible Mandatory Access Control (MAC) that restricts the access of users and processes to files and directories.

#### Least Privilege Principle

**Definition 3.1.7** (Least Privilege Principle) The principle of least privilege (POLP) is an important concept in computer security, promoting minimal user profile privileges on computers based on users' job necessity.

This principle states that a user should have only the minimum privileges required to perform their job. This principle is applied throughout linux, and also in SELinux.

You can check if **SELinux** is enabled by running

```
1 | $ sestatus
```

If SELinux is enabled, you can check the context of a file or a directory using

```
1 | $ ls -lZ
```

However, if SELinux is not enabled, it will show a ? in the context.

If SELinux is enabled, you can set the context of a file or a directory using the chcon command.

#### RBAC Items

**Definition 3.1.8 (Role-Based Access Control (RBAC))** Role-Based Access Control (RBAC) is a policy-neutral access control mechanism defined around roles and privileges. The components of RBAC such as role-permissions, user-role, and role-role relationships make it simple to perform user assignments.

SELinux uses the concept of RBAC to control the access of users and processes to files and directories.

There are four components in the SELinux context that are used to control the access of users and processes to files and directories, as shown in Figure 3.6.

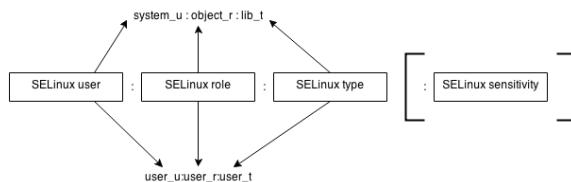


Figure 3.6: SELinux Context

- ▶ **User:** The user who is trying to access the file or directory.
- ▶ **Role:** The role of the user, which defines the permissions of the user.
- ▶ **Type:** The type of the file or directory.
- ▶ **Domain:** The domain <sup>9</sup> of the process trying to access the file or directory.

9: or type or sensitivity

## Modes of SELinux

- ▶ **Enforcing:** In this mode, SELinux is enabled and actively enforcing the security policies.
- ▶ **Permissive:** In this mode, SELinux is enabled but not enforcing the security policies. It logs the violations but does not block them.

- **Disabled:** In this mode, SELinux is disabled and not enforcing any security policies.

You can change the mode of SELinux by editing the `/etc/selinux/config` file.

```
1 | $ sudo vim /etc/selinux/config
```

### Tools for SELinux

- **`sestatus`:** Check the status of SELinux.
- **`semamange`:** Manage the SELinux policy.
- **`restorecon`:** Restore the context of files and directories.

## 3.1.12 Network Tools

There are a lot of tools in GNU/Linux used for managing, configuring, and troubleshooting networks. Some of the important tools are listed in Table 3.3.

**Table 3.3:** Network Tools

Tool	Description
<code>ip</code>	Show / manipulate routing, devices, policy routing and tunnels
<code>ping</code>	To see if the remote machine is up
<code>traceroute</code>	Diagnostics the hop timings to the remote machine
<code>nslookup</code>	Ask for conversion of IP address to name
<code>dig</code>	DNS lookup utility
<code>netstat</code>	Print network connections
<code>mxtoolbox</code>	Public accessibility of your server
<code>whois</code>	Information about the domain
<code>nmap</code>	Network port scanner
<code>wireshark</code>	Network protocol analyzer and packet sniffer

### ip

To find out the private IP address of the NICs of your system, you can run the `ip addr` command.

10: `ip a` also works.

```
1 $ ip addr
2 1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc
   noqueue state UNKNOWN group default qlen
   1000
3     link/loopback 00:00:00:00:00:00 brd
   00:00:00:00:00:00
4     inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
5     inet6 ::1/128 scope host noprefixroute
       valid_lft forever preferred_lft forever
6 2: eno1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu
   1500 qdisc fq_codel state UP group
   default qlen 1000
7     link/ether 1c:1b:0d:e1:5d:61 brd ff:ff:ff:
   ff:ff:ff
8     altname enp3s0
9     inet 192.168.0.109/24 brd 192.168.0.255
   scope global dynamic eno1
10    valid_lft 7046sec preferred_lft 7046sec
11    inet6 fe80::68e2:97e0:38ec:4abc/64 scope
   link noprefixroute
12    valid_lft forever preferred_lft forever
13
14
```

Here you can see there are two interfaces, **lo** and **eno1**. The **lo** interface is the loopback interface, and the **eno1** interface is the actual network interface. The IP address of the **lo** interface is usually always **127.0.0.1**. This address is used to refer to the same system in terms of IP address without knowing the actual private IP of the system in the LAN.

The IP address of the **eno1** interface is the private IP address allocated by your router. This is not your public IP address, which is the address of your router on the internet. Usually public IPs are statically assigned by ISPs and are not changed often. It is configured in your router.

Private IPs however often needs to be assigned dynamically since devices can connect and disconnect

from the network at any time. This is done by the DHCP server in your router.

**Remark 3.1.2** The NIC name can be different in different systems. For a ethernet connection, it is usually `eno1` or `eth0` which is the legacy name. For a wifi NIC, it is usually `wlan0`.

**Remark 3.1.3** Earlier the tool used to check the network status was `ifconfig`. However, this tool is deprecated now and should not be used. The new tool to check the network status is `ip`.

## ping

11: Internet Control Message Protocol (ICMP) is a supporting protocol in the Internet protocol suite. It is used by network devices, including routers, to send error messages and operational information indicating success or failure when communicating with another IP address.

**Remark 3.1.4** Only a positive response from the server indicates that the server is up and running. A negative response does not necessarily mean that the server is down. Servers can be configured to not respond to ICMP packets.

```

1 $ ping -c 4 google.com # Send 4 ICMP packets
   to google.com
2 PING google.com (172.217.163.206) 56(84) bytes
   of data.
3 64 bytes from maa05s06-in-f14.1e100.net
   (172.217.163.206): icmp_seq=1 ttl=114 time
   =45.6 ms
4 64 bytes from maa05s06-in-f14.1e100.net
   (172.217.163.206): icmp_seq=2 ttl=114 time
   =45.4 ms
5 64 bytes from maa05s06-in-f14.1e100.net
   (172.217.163.206): icmp_seq=3 ttl=114 time

```

```

1      =45.3 ms
6 64 bytes from maa05s06-in-f14.1e100.net
    (172.217.163.206): icmp_seq=4 ttl=114 time
    =45.8 ms
7
8 --- google.com ping statistics ---
9 4 packets transmitted, 4 received, 0% packet
    loss, time 3004ms
10 rtt min/avg/max/mdev =
    45.316/45.524/45.791/0.181 ms

```

The response of the ping command shows the time taken for the packet to reach the server and also the resolved IP address of the server.

### **nslookup**

Another tool to lookup the associated IP address of a domain name is the nslookup command.

```

1 $ nslookup google.com
2 Server:      192.168.0.1
3 Address:     192.168.0.1#53
4
5 Non-authoritative answer:
6 Name:  google.com
7 Address: 172.217.163.206
8 Name:  google.com
9 Address: 2404:6800:4007:810::200e

```

Here you can see the resolved IP address of the domain is 172.217.163.206. If you copy this IP address and paste it in your browser, you can see that the website of google opens up. The second address returned is the IPv6 IP Address.

The first lines mentioning the **Server** is the **DNS Server** which returned the resolution of the IP address from the queried domain name.

**Remark 3.1.5** Notice that the DNS Server mentioned in the above output is actually a private IP. This is the IP address of the router in the LAN which acts as the DNS Server cache. However if you type the domain of a website which you have not visited, or have visited long ago into nslookup, then the DNS Server mentioned will be the public address of the DNS Server, which might be your ISP's DNS Server, or some other public DNS Server.

You can also use **mxtoolbox** to check the IP address of your server from the public internet.

### dig

Another tool to lookup the associated IP address of a domain name is the **dig** command. It can also reverse lookup the IP address to find the associated domain name.

```

1 $ dig google.com
2
3 ; <>> DiG 9.18.27 <>> google.com
4 ;; global options: +cmd
5 ;; Got answer:
6 ;; ->>HEADER<<- opcode: QUERY, status: NOERROR
7 , id: 31350
8 ;; flags: qr rd ra; QUERY: 1, ANSWER: 1,
9 AUTHORITY: 4, ADDITIONAL: 9
10
11 ;; OPT PSEUDOSECTION:
12 ; EDNS: version: 0, flags:; udp: 4096
13 ; COOKIE: 3
14 e5ff6a57c0fe2b3b5ce91b3666ae859ec9b6471261cecef
15 (good)
16
17 ;; QUESTION SECTION:
18 ;google.com.           IN  A
19
20 ;; ANSWER SECTION:
```

```
16 google.com.      50  IN  A   172.217.163.206
17
18 ;; AUTHORITY SECTION:
19 google.com.      162911  IN  NS  ns1.google.com
20 .
21 google.com.      162911  IN  NS  ns3.google.com
22 .
23 google.com.      162911  IN  NS  ns4.google.com
24 .
25 google.com.      162911  IN  NS  ns2.google.com
26 .
27
28 ;; ADDITIONAL SECTION:
29 ns2.google.com.    163913  IN  A   216.239.34.10
30 ns4.google.com.    163913  IN  A   216.239.38.10
31 ns3.google.com.    337398  IN  A   216.239.36.10
32 ns1.google.com.    340398  IN  A   216.239.32.10
33 ns2.google.com.    163913  IN  AAAA 2001:4860:4802:34::a
34 ns4.google.com.    163913  IN  AAAA 2001:4860:4802:38::a
35 ns3.google.com.    2787    IN  AAAA 2001:4860:4802:36::a
36 ns1.google.com.    158183  IN  AAAA 2001:4860:4802:32::a
37
38
39 ;; Query time: 3 msec
40 ;; SERVER: 192.168.0.1#53(192.168.0.1) (UDP)
41 ;; WHEN: Thu Jun 13 18:18:52 IST 2024
42 ;; MSG SIZE rcvd: 331
```

And we can then feed the IP address to dig again, to find the domain name associated with the IP address.

```
1 $ dig -x 172.217.163.206
2
```

```
3 ; <>> DiG 9.18.27 <>> -x 172.217.163.206
4 ;; global options: +cmd
5 ;; Got answer:
6 ;; ->>HEADER<- opcode: QUERY, status: NOERROR
7 ;;; flags: qr rd ra; QUERY: 1, ANSWER: 1,
8 ;;; AUTHORITY: 4, ADDITIONAL: 9
9
10;; OPT PSEUDOSECTION:
11; EDNS: version: 0, flags:; udp: 4096
12; COOKIE: 78
13 ;206.163.217.172.in-addr.arpa. IN PTR
14
15;; ANSWER SECTION:
16206.163.217.172.in-addr.arpa. 83966 IN PTR
17 ;maa05s06.in-f14.1e100.net.
18
19;; AUTHORITY SECTION:
20217.172.in-addr.arpa. 78207 IN NS ns4.
21 ;google.com.
22217.172.in-addr.arpa. 78207 IN NS ns2.
23 ;google.com.
24217.172.in-addr.arpa. 78207 IN NS ns1.
25 ;google.com.
26217.172.in-addr.arpa. 78207 IN NS ns3.
27 ;google.com.
28
29;; ADDITIONAL SECTION:
30ns1.google.com. 340332 IN A
31 ;216.239.32.10
32ns2.google.com. 163847 IN A
33 ;216.239.34.10
34ns3.google.com. 337332 IN A
35 ;216.239.36.10
36ns4.google.com. 163847 IN A
37 ;216.239.38.10
38ns1.google.com. 158117 IN AAAA
39 ;2001:4860:4802:32::a
```

```

30 ns2.google.com.      163847 IN  AAAA
     2001:4860:4802:34::a
31 ns3.google.com.      2721   IN  AAAA
     2001:4860:4802:36::a
32 ns4.google.com.      163847 IN  AAAA
     2001:4860:4802:38::a
33
34 ;; Query time: 3 msec
35 ;; SERVER: 192.168.0.1#53(192.168.0.1) (UDP)
36 ;; WHEN: Thu Jun 13 18:19:58 IST 2024
37 ;; MSG SIZE  rcvd: 382

```

Note that the answer we got after running `google.com` through `dig` and then through `dig -x` (maa05s06-in-f14.1e100.net) is different from the original domain name.

This is because the domain name is resolved to an IP address, and then the IP address is resolved to a different domain name. This is because the domain name is actually an alias to the canonical name.

**Remark 3.1.6** The IP address you would get by running `dig` or `nslookup` on `google` would be different from the IP address you get when using `mxtoolbox`. This is because `google` is a large company and they have multiple servers which are load balanced. So someone in India might get a different IP address compared to someone in the US.

To get the output of `dig` in a more readable and concise format, you can use the `+short` or `+noall` option.

```

1 $ dig +noall +answer google.com
2 google.com.      244 IN  A   172.217.163.206

```

## netstat

The `netstat` command is used to print network connections, routing tables, interface statistics, masquerade connections, and multicast memberships.

It is useful to find what connections are open on your system, and what ports are being used by which applications.

```
1 $ netstat | head
2 Active Internet connections (w/o servers)
3 Proto Recv-Q Send-Q Local Address
   Foreign Address      State
4 tcp      0      0 rex:53584
   24.224.186.35.bc.:https TIME_WAIT
5 tcp      0      0 rex:56602
   24.224.186.35.bc.:https TIME_WAIT
6 tcp      0      0 localhost:5037
   localhost:43267      TIME_WAIT
7 tcp      0      0 localhost:5037
   localhost:46497      TIME_WAIT
8 tcp      0      0 rex:35198
   24.224.186.35.bc.:https TIME_WAIT
9 tcp      0      0 rex:44302
   24.224.186.35.bc.:https TIME_WAIT
10 tcp     0      0 localhost:5037
   localhost:55529      TIME_WAIT
11 tcp     0      0 localhost:5037
   localhost:38005      TIME_WAIT
```

## 3.2 SSH

### 3.2.1 What is SSH?

The Secure Shell (SSH) Protocol is a protocol for secure communication between two computers over a compromised or untrusted network.<sup>12</sup> SSH uses encryption and authentication to secure the communication between the two computers.

12: like the internet.

SSH is now the ubiquitous protocol for secure remote access to servers, and is used by system administrators all over the world to manage their servers.

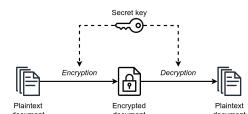
SSH lets a user of a computer to log into the computer from another computer over the network, to execute any command in the terminal that they have access to.

SSH can also be used to transfer files between two computers using the `scp` command.

### 3.2.2 History

It was initially developed by Tatu Ylönen in 1995 as a replacement for the insecure Telnet and FTP protocols when he found that someone had installed a packet sniffer on the server of his university.

There are multiple implementations of the SSH protocol, the most popular being OpenSSH, developed by the OpenBSD project. This is the implementation that is used in most of the linux distributions as well.

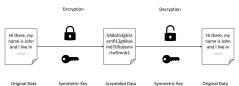


**Figure 3.7:** Symmetric Encryption

### 3.2.3 How does SSH work?

SSH works by using symmetric and asymmetric encryption. The data packets sent over the network are encrypted, usually using AES symmetric encryption. This ensures that even if the data packets are intercepted by a man-in-the-middle attacker, they cannot be read since they are encrypted.

To login into a remote server, all you need to do is provide the username and the IP address or the domain name of the server to the `ssh` command.



```
1 | $ ssh username@ipaddress
```

OR

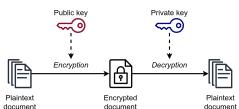
**Figure 3.8:** Symmetric Encryption

```
1 | $ ssh username@domainname
```

SSH allows user to login to a remote server using their username and password, but this is not encouraged since it lets the user to be vulnerable to brute-force attacks.

Another way to authenticate is by using public-private key pairs.

### 3.2.4 Key-based Authentication



**Figure 3.9:** Asymmetric Encryption

One of the most powerful features of SSH is its ability to use public-private key pairs for authentication. In our course, we emphasize the importance of this method. Instead of relying on passwords, which can be vulnerable to brute-force attacks, a pair of cryptographic keys is used. The public key is stored on the server, while the private key is kept secure on your local machine. This ensures a highly secure and convenient way of accessing remote

servers without the need for constantly entering passwords.

### 3.2.5 Configuring your SSH keys

For this course, it is a must for you to not only create, but also understand SSH keys. Let us quickly see how to create a ssh key-pair which can be used to login into a remote server.

We need to use the `ssh-keygen` command to create a new public-private key pair.

```
1 $ ssh-keygen  
2 Generating public/private ed25519 key pair.  
3 Enter file in which to save the key (/home/  
    test1/.ssh/id_ed25519):
```

Here you have to either simply press enter to continue with the default location, or you can also type a custom location where you want to save the key. If this is your first time creating keys, it is recommended to use the default location.

**Remark 3.2.1** There are multiple algorithms that can be used to generate a key pair. The most common ones are RSA, DSA, and ED25519. The ED25519 algorithm is the new default algorithm used by OpenSSH since it is shorter yet more secure than RSA. If you have an outdated version of OpenSSH, you might get the default RSA algorithm. To change the algorithm, you can use the `-t` flag along with the `ssh-keygen` command.

```
1 $ ssh-keygen -t rsa  
will create a RSA key pair and using  
1 $ ssh-keygen -t ed25519  
will create a ED25519 key pair.
```

Next, it will ask you to enter a passphrase. You can enter a passphrase for added security, or you can simply press enter to continue without a passphrase. If you do add a passphrase, you will have to always enter the passphrase whenever you use the key. We can continue without a passphrase for now by pressing enter.

```
1 Enter passphrase (empty for no passphrase):  
2 Enter same passphrase again:  
3  
4 Your identification has been saved in /home/  
    username/.ssh/id_ed25519  
5 Your public key has been saved in /home/  
    username/.ssh/id_ed25519.pub  
6 The key fingerprint is:  
7 SHA256:  
    n4ounQd6v9uWXAtMyyq7CdncMsh1Zuac5jesWXrndeA  
    test1@rex  
8 The key's randomart image is:  
9 +-- [ED25519 256] --+  
10 |  
11 |  
12 |  
13 |       .  
14 |       . S+ . . |  
15 |   . *.0 o=.... . |  
16 |   =o=o*+++.E . |  
17 |   o.=Bo*0 o. . |  
18 |   ***XB.+.  
19 +--- [SHA256] -----+
```

Our key pair has been generated. The private key is stored in `/home/username/.ssh/id_ed25519` and the public key is stored in `/home/username/.ssh/id_ed25519.pub`.

Make sure to **never** share your private key with anyone. Ideally, you dont even need to see the private key yourself. You should only share the public key with the server you want to login to.

### 3.2.6 Sharing your public key

#### ssh-copy-id

Finally, to share the public key with the server, there are usually multiple ways. If the server allows you to login using a password, you can simply use the `ssh-copy-id` command. This command will take your username and password to login to the server, and then copy the public key which you provide to the server.

```
1 $ ssh-copy-id -i /key/to/public/key  
      username@ipaddress
```

**Remark 3.2.2** The `-i` flag is used to specify the path to the public key. You can drop the `.pub` from the path as well (making it the path to the private key), since `ssh-copy-id` will automatically look for the public key. However, this flag is not required if you are using the default location. This is why using the default location is recommended for beginners. The simplified syntax then becomes

```
1 $ ssh-copy-id username@ipaddress
```

The same applies for logging into the server using the `ssh` command.

#### manual install

However, most servers do not allow password login at all, since it defeats the purpose of using a public-private key pair. In such cases, you need to somehow copy the public key to the server.

If you have physical access to the server, you can simply copy the public key to the server in the `~/.ssh/authorized_keys` file of the server.

```

1 | $ file ~/someoneskey.pub
2 | ~/someoneskey.pub: OpenSSH ED25519 public key
3 | $ cat ~/someoneskey.pub >> ~/.ssh/
   |     authorized_keys

```

**Remark 3.2.3** Make sure to use the `>>` operator and not the `>` operator. The `>>` operator appends the contents of the file to the end of the file, while the `>` operator overwrites the file, we do not want that.

### System Commands Course

However, in case of our course, you do not have access to the server as well. To submit your **public key**, you have to login into the website <https://se2001.ds.study.iitm.ac.in/passwordless> using your institute credentials, and then submit your public key in the form provided.

You can print out the contents of the public key using the `cat` command and copy the contents into the form.

```

1 | [test1@rex ~]$ cat .ssh/id_ed25519.pub
2 | ssh-ed25519
   | AAAAC3NzaC1lZDI1NTE5AAAIIDxh5EuvzQkGvsq1MQW3r0kY
   | +wyo+2d6Y5CSqNGlLs2a test1@rex

```

You should copy the entire contents of the file, including your username and hostname.

### 3.2.7 How to login to a remote server

You can then login into the server using the `ssh` command.

```

1 | $ ssh rollnumber@se2001.ds.study.iitm.ac.in

```

OR, if not using the default location of key

```
1 $ ssh -i /path/to/private/key
      rollnumber@se2001.ds.study.iitm.ac.in
```

If successful, you will be logged into the server and the prompt will change to the server's prompt.

```
1 [test1@rex ~]$ ssh 29f1001234@se2001.ds.study.
      iitm.ac.in
2 Last login: Mon Jun  3 07:43:22 2024 from
      192.168.2.3
3 29f1001234@se2001:~$
```

Notice that the prompt has changed from `test1@rex` which was the prompt of your local machine, to `rollnumber@se2001` which is the prompt of the server.

### 3.2.8 Call an exorcist, there's a daemon in my computer

What is `sshd`? It is a daemon.

**Definition 3.2.1** (Daemon) In multitasking computer operating systems, a daemon is a computer program that runs as a background process, rather than being under the direct control of an interactive user.

There are many daemons running in your computer. You can use `systemctl status` to see the loaded and active daemons in your computer.

```
1 $ systemctl status
2 * rex
3     State: running
```

```

4      Units: 419 loaded (incl. loaded aliases)
5      Jobs: 0 queued
6      Failed: 0 units
7      Since: Thu 2024-06-13 12:55:42 IST; 7h ago
8      systemd: 255.6-1-arch
9      CGroup: /
10         |-init.scope
11             |- /usr/lib/systemd/systemd --
12                 switched-root --system --deserialize=43
13                     |-system.slice
14                         |-NetworkManager.service
15                             |-547 /usr/bin/NetworkManager
16                             --no-daemon
17                                 |-adb.service
18                                     |-558 adb -L tcp:5037 fork-
19                                         server server --reply-fd 4
20                                         |-avahi-daemon.service
21                                         |-550 "avahi-daemon: running [
22                                             rex.local]"
23                                         |-557 "avahi-daemon: chroot
24                                         helper"
25                                         |-cronie.service
26                                         |-621 /usr/sbin/crond -n
27                                         |-cups.service
28                                         |-629 /usr/bin/cupsd -l
29                                         |-dbus-broker.service
30                                         |-545 /usr/bin/dbus-broker-
31                                         launch --scope system --audit

```

Here you can see some of the important daemons running, such as NetworkManager which is used to manage the network connections, cronie which is used to run scheduled tasks, cups which is used to manage printers, etc.

### sshd

sshd is the daemon that runs on the server and listens to any incoming SSH connections. It is the daemon that lets you login into the server using the SSH protocol.

Your own system might not be running the sshd daemon, since you are not running a server. However, you can check if the sshd daemon is running using the systemctl command.

```

1 $ systemctl status sshd
2 * sshd.service - OpenSSH Daemon
3     Loaded: loaded (/usr/lib/systemd/system/
4         sshd.service; disabled; preset: disabled)
4     Active: inactive (dead)
```

Here you can see that the sshd daemon is currently inactive. This is because I am not running a server and don't usually login remotely to my system. However, the output of the same command would be something like as shown below if it is enabled on your system.

```

1 $ systemctl status sshd
2 * sshd.service - OpenSSH Daemon
3     Loaded: loaded (/usr/lib/systemd/system/
4         sshd.service; disabled; preset: disabled)
4     Active: active (running) since Thu
5         2024-06-13 19:48:44 IST; 12min ago
5     Main PID: 3583344 (sshd)
6         Tasks: 1 (limit: 9287)
7         Memory: 2.1M (peak: 2.3M)
8         CPU: 8ms
9         CGroup: /system.slice/sshd.service
10            '-3583344 "sshd: /usr/bin/sshd -D
11              [listener] 0 of 10-100 startups"
12
12 Jun 13 19:48:44 rex systemd[1]: Started
13   OpenSSH Daemon.
13 Jun 13 19:48:45 rex sshd[3583344]: Server
14     listening on 0.0.0.0 port 22.
14 Jun 13 19:48:45 rex sshd[3583344]: Server
15     listening on :: port 22.
```

If we run the same command on the server, we can see that it is running. However, we wont be

able to read the logs of the server, since we are not authorized.

I have set the LC\_ALL environment variable to the locale C while generating the above outputs to prevent latex errors. Ideally if you run the command, you will see a prettier unicode output.

```

1 $ ssh username@se2001.ds.study.iitm.ac.in
2 username@se2001:~$ systemctl status sshd
3 * ssh.service - OpenBSD Secure Shell server
4     Loaded: loaded (/lib/systemd/system/sshd.
5         service; enabled; vendor preset: enabled)
6             Active: active (running) since Thu
7                 2024-06-13 12:32:47 UTC; 1h 57min ago
8                   Docs: man:sshd(8)
9                         man:sshd_config(5)
10                  Process: 732 ExecStartPre=/usr/sbin/sshd -
11                      t (code=exited, status=0/SUCCESS)
12                  Main PID: 745 (sshd)
13                     Tasks: 1 (limit: 4557)
14                   Memory: 22.0M
15                     CPU: 8.769s
16                   CGroup: /system.slice/sshd.service
17                           '-745 "sshd: /usr/sbin/sshd -D [
18                             listener] 0 of 10-100 startups"
19
20 Warning: some journal files were not opened
21 due to insufficient permissions.

```

**Remark 3.2.4** Notice that there are some differences in the output when run from my local system and from the system commands server. Such as, the name of the service is ssh on the server, while it is sshd on my local system. Also the full name is **OpenBSD Secure Shell server** on the server, while it is **OpenSSH Daemon** on my local system. The path of the service file is also different. This is because the server is running a ubuntu distribution whereas my local system runs an arch distribution. They have different packages for ssh, and hence the differences.

### 3.2.9 SCP

**scp** is a command used to copy files between remote servers. It uses the **ssh** protocol to copy files in an encrypted manner over the network.

The syntax of the **scp** command is similar to the **cp** command.

```
1 $ scp username@ipaddress:/path/to/file /path/  
    to/destination
```

This will copy a file from the remote server to your local machine.

```
1 $ scp /path/to/file username@ipaddress:/path/  
    to/destination
```

This will copy a file from your local machine to the remote server.



# Process Management

## 4.1 What is sleep?

`sleep` is a command that is used to delay the execution of a process for a specified amount of time. `sleep` itself is a no-op command,\* but it takes a variable amount of time to execute, depending on the argument of the command. This is useful when you want to delay the execution of another command or chain of commands by a certain amount of time.

### 4.1.1 Example

```
1 $ sleep 5
2 $ echo "Hello, World!"
3 Hello, World!
```

### 4.1.2 Scripting with sleep

If you run the above snippet, you will see that the output is delayed by 5 seconds. Moreover, the prompt itself will not be available for 5 seconds, as the shell is busy with executing the `sleep` command. To run the entire snippet as one process, simply put the two commands on separate lines of a file (say, `hello.sh`), and run the file as a script.

```
1 $ cat hello.sh
2 sleep 5
3 echo "Hello, World!"
4 $ bash hello.sh
5 Hello, World!
```

\* NO-OP stands for No Operation. It is a command that does nothing. More reading on NO-OP can be found [here](#).

We will be using `sleep` in the examples throughout this chapter to demonstrate process management since it is a simple command that can be used to quickly spawn an idempotent process for any arbitrary amount of time.

### 4.1.3 Syntax and Synopsis

```
1 | sleep NUMBER[SUFFIX]...
```

Here the `NUMBER` is the amount of time to sleep. The `SUFFIX` can be `s` for seconds, `m` for minutes, `h` for hours, and `d` for days.

## 4.2 Different ways of running a process

### 4.2.1 What are processes?

**Definition 4.2.1** (Process) A process is an instance of a program that is being executed. It contains the program code and its current activity. Depending on the operating system (OS), a process may be made up of multiple threads of execution that execute instructions concurrently. Several processes may be associated with the same program; for example, opening up several instances of the same program often means more than one process is being executed. Each process has its own ‘process id’ or **PID** to uniquely identify it.

Whenever we run an application, or even a command on the linux shell, it spawns a process. Processes are always created by an already existing process<sup>1</sup> This creates a tree-like structure of processes, where each process has a parent process and can have multiple child processes. When the parent of a process dies, the child processes are adopted by the **init** process. **init** is thus the root of the process tree.

1: Other than the very first process, which is always the **init** process. In most distributions, this is done by **systemd**, which is an init system that does a lot of other things as well. You can learn more about systemd and what all it does [here](#).

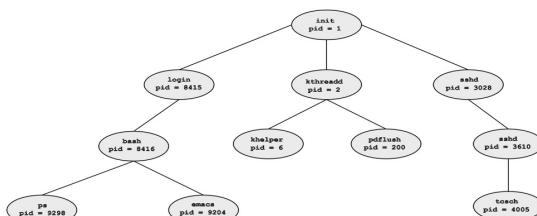


Figure 4.1: Example of a process tree

### 4.2.2 Process Creation

In linux systems, processes are managed by the kernel. The kernel is responsible for creating, scheduling, and destroying processes. The user can interact with the kernel using system calls to create, manage, and destroy processes. Creating processes is simple, and can be done using the **fork()** system call. This is used when any process wants to create a new process.

To simply create a new process for a command, we can simply type in the command and press enter. This will not only fork a new process from the terminal or terminal emulator as the parent process, but also tie the standard input, standard output, and standard error of the child process to the terminal or terminal emulator.<sup>2</sup>

2: Standard Input, Output, and Error are the default streams that are used by the shell to interact with the user.<sup>1</sup>

Standard Input is used to take input from the user, Standard Output is used to display output to the user, and Standard Error is used to display errors to the user. We will cover these in details in the next chapter.

```
$ sleep 5
```

This will create a new process that will sleep for 5 seconds.

Remember that each process has a unique process id (PID). Each process also has a parent process id (PPID), which is the PID of the parent process. If a process is created by the shell, the shell will be the parent process. If the shell's process is killed, the child process will also be killed, as the child process is owned by the shell.

### 4.2.3 Process Ownership

If you are using a linux operating system with a GUI server (X or Wayland), try the following to understand how process ownership works.

Open two terminals, in the first one, run echo \$\$ to see the process ID of that shell. It should print out a random string of digits, that is the PID of the shell. Then run a GUI application, such as **firefox**.

<sup>3</sup> This will block your terminal and open a new window of firefox.

```
1 $ echo $$  
2 2277503  
3 $ firefox
```

Now in the other terminal, which is free, run pgrep firefox<sup>4</sup> It should print out another random string of digits, it is the PID of firefox.

Now you can use the following command to find the parent process's process ID (PPID) to verify it is the same as the output of \$\$ in the first terminal.

```
1 $ pgrep firefox  
2 2278276  
3 $ ps -efj | awk '$2==2278276;NR==1'  
4 UID          PID      PPID      PGID      SID  C  
      STIME     TTY          TIME CMD  
5 sayan      2278276  2277503  2278276  2277503 12  
           16:59 pts/5    00:00:03 /usr/lib/firefox/  
           firefox
```

Here we can see that the PPID of firefox is the PID of the shell.

Note that the second command should put the PID of firefox, which we got from the previous command. This can also be done in a single command which you can directly copy and paste in your terminal.

```
1 $ ps -efj | awk "\$2==$(pgrep firefox);NR==1"  
2 UID          PID      PPID      PGID      SID  C  
      STIME     TTY          TIME CMD
```

3: Make sure you are running something that is not running already.

4: or whatever was your process's name

```
3 sayan    2278276 2277503 2278276 2277503  1
      16:59 pts/5    00:00:04 /usr/lib/firefox/
      firefox
```

Now, what happens if we kill the parent process?  
To kill a process all we need to use is use the `kill` command with the PID of the process.

```
1 | $ kill -9 2277503
```

The `-9` flag is used to send a **SIGKILL** signal to the process. This signal is used to kill a process immediately. We will cover signals later.

If you have been following along, you will see that both the terminal and firefox dissapear from your screen. You will also notice that if you run the same command to print the PID and PPID of firefox, it does not show anything. This is because the process is killed and the process tree is destroyed, so even firefox, being the child of the shell process, is killed.

#### 4.2.4 Don't kill my children

However, there are also ways to create a new process in the background. The easiest way to do this is to append an ampersand (`&`) to the end of the command. This is a shell syntax that tells the shell to fork the command as a child process and run it in the background. What this means is the shell will not wait for the command to finish, and will return the prompt to the user immediately. However, the standard output and standard error may still be tied to the terminal or terminal emulator. So if the process writes something to the standard output or standard error, it will be displayed on the terminal. Furthermore, the process is still owned by the shell, and if the shell is killed, the process's parent will be changed to the `init` process.

Lets try the same exercise as earlier, but now with the `&` at the end.

Open two terminals, and in the first one, execute the following command.

```

1 $ echo $$
2 2400520
3 $ firefox &
4 [1] 2401297
5 $ echo "hello"
6 hello
7 $
8 ATTENTION: default value of option
     mesa_glthread overridden by environment.
9 $
```

You can observe that the `firefox` window opens up similar to last time, but now the prompt returns immediately. You can also see that the output of the `echo` command is displayed on the terminal.

If you try to perform some operations in the browser, it may also print some messages to the terminal screen, even though it is not waiting for the command to finish. The "ATTENTION" message is an example of this.

Also observe that as soon as we launched `firefox`, it printed out two numbers, [1] and 2401297. The number in the square brackets is the job id of the process, and the number after that is the PID of the process. So now we dont even need to use `pgrep` to find the PID of the process.

Now in the other terminal, run the following command.

```

1 $ ps -efj | awk "\$2==$(pgrep firefox);NR==1"
2   UID          PID      PPID      PGID      SID  C
      STIME    TTY          TIME CMD
3 sayan      2401297  2400520  2401297  2400520  3
      17:13 pts/5    00:00:08 /usr/lib/firefox/
      firefox
```

Still we can see that the PPID of firefox is the PID of the shell.

Now, if we kill the parent process, the child process will be adopted by the init process, and will continue to run.

```
1 | $ kill -9 2400520
```

If you re-run the command to print the PID and PPID of firefox, you will see that the PPID of firefox is now set to 1, which is the PID of the **init** command.

```
1 | $ ps -efj | awk "\$2==$(pgrep firefox);NR==1"
2 |          UID      PID      PPID      PGID      SID   C
3 |          STIME    TTY      TIME     CMD
4 | sayan      2401297      1 2401297 2400520  3
5 |          17:13 ?      00:00:09 /usr/lib/firefox/
6 |          firefox
```

You can also see that the TTY column is now set to ?, which means that the process is no longer tied to the terminal.

However, if instead of killing the parent process using the **SIGKILL** signal, if you sent the **SIGHUP** signal to the parent, the child process will still be terminated, as it will propagate the hangup signal to the child process.

#### 4.2.5 Setsid

So how do we start a process directly in a way that it is not tied to the terminal? Many times we would require to start a process in the background to run asynchronously, but not always do we want to see the output of the process in the terminal from where we launched it. We may also want the process to be owned by the **init** process from the get go.

To do this, we can use the `setsid` command. This command is used to run a command in a new session. This will create a new process group and set the PPID of the process to the `init` process. The TTY will also be set to ?.

Lets try the same exercise with the `setsid` command. Open two terminals, in one of them, run the following command.

```
1 $ echo $$  
2 2453741  
3 $ setsid -f firefox  
4 $
```

Observe that firefox will open up, but the prompt will return immediately.

In another terminal, run the following command.

```
1 $ ps -efj | awk "\$2==$(pgrep firefox);NR==1"  
2 UID          PID      PPID      PGID      SID  C  
   STIME       TTY        TIME CMD  
3 sayan      2454452      1 2454452 2454452  2  
    17:19 ?      00:00:07 /usr/lib/firefox/  
      firefox
```

Observe that even without killing the parent process, the PPID of firefox is already set to 1, which is the PID of the `init` process. So the process will not be killed if the shell is killed.

This is called a hang-up signal. We can still artificially send the **SIGHUP** signal, which tells firefox that its parent has stopped by using the `kill -1` command.

```
1 $ kill -1 2454452
```

This will still close firefox, even though the parent process (`init`) didn't actually get killed.

### 4.2.6 Nohup

If you do not want to give up the ownership of a child process, and also don't really need to get the prompt back, but you do not want to see the output of the command in your terminal. You can use the `nohup` command followed by the command you want to run. It will still be tied to the terminal, and you can use `Ctrl+C` to stop it, `Ctrl+Z` to pause it, etc. The prompt will also be blocked till the process runs. However, the input given to the terminal will not be sent to the process, and the output of the process will not be shown on the terminal. Instead, the output will be saved in a file named `nohup.out` in the current directory.

- 5: We will cover redirection operators in the next chapter.

However, this is different from simply running the command with a redirection operator (`>`) at the end,<sup>5</sup> because the `nohup` command also makes the process immune to the hang-up signal.

**Exercise 4.2.1** Try the same exercise as before, but this time use the `nohup` to run `firefox`, then in another terminal, find the PID and PPID of `firefox`. Then try to kill the parent process and see if `firefox` dies or not.

### 4.2.7 coproc

The `coproc` command is used to run a command in the background and tie the standard input and standard output of the command to a file descriptor. This is useful when you want to run a command in the background, but still want to interact with it using the shell. This creates a two way pipe between the shell and the command.

## Syntax

```
1 | $ coproc [NAME] command [redirections]
```

This creates a coprocess named NAME and runs the command in the background. If the NAME is not provided, the default name is COPROC.

However, the recommended way to use coproc is to use it in a subshell, so that the file descriptors are automatically closed when the subshell exits.

```
1 | $ coproc [NAME] { command; }
```

coproc can execute simple commands or compound commands. For simple commands, name is not possible to be specified. Compound commands like loops or conditionals can be executed using coproc in a subshell.

The name that is set becomes a array variable in the shell, and can be used to access the file descriptors for stdin, stdout, and stderr.

For example, to provide input to the command, you can use echo and redirection operators to write to the file descriptor.

Similarly you can use the read command to read from the file descriptor.

```
1 | $ coproc BC { bc -l; }
2 | $ jobs
3 | [1]+  Running                      coproc BC { bc -
4 |       l; } &
5 | $ echo 22/7 >&"${BC[1]}"
6 | $ read output <&"${BC[0]}"
7 | $ echo $output
7 | 3.14285714285714285714
```

This uses concepts from redirection and shell variables, which we will cover in later weeks.

### 4.2.8 at and cron

Processes can also be scheduled to be launched at a later time. This is usually done using **cron** or the **at** command. We will cover these in depth later.

### 4.2.9 GNU parallel

GNU parallel is a shell tool for executing jobs in parallel using one or more computers. A job can be a single command or a small script that has to be run for each of the lines in the input. The typical input is a list of files, a list of hosts, a list of users, a list of URLs, or a list of tables. A job can also be a command that reads from a pipe. GNU parallel can then split the input into blocks and pipe a block into each command in parallel.

### 4.2.10 systemd services

Finally, the best way to run a background process or a daemon is to use **systemd** services. **systemd** is an init system that is used by most modern linux distributions. You can create a service file declaring the process, the command line arguments, the environment variables, and the user and group that the process should run as. You can also specify if the process should be restarted if it crashes, or if it should be started at boot time.

## 4.3 Process Management

### 4.3.1 Disown

Disown is a shell builtin command that is used to remove a job from the shell's job table. This is useful when you have started a process in the background and you want to remove it from the shell's job table, so that it is not killed when the shell is killed. What it means is that if the parent process receives a hang-up signal, it will not propagate it to the child job if it is removed from the job table. This is applicable only for processes started from a shell.

Open two terminals, in one, open firefox in background using the &

```
1 $ firefox &
2 $
```

and then in the other terminal, run the following command.

```
1 $ ps -efj | awk "\$2==$(pgrep firefox);NR==1"
2   UID          PID      PPID      PGID      SID  C
3           STIME    TTY        TIME CMD
3 sayan      3216429  3215856  3216429  3215856 69
4       18:45 pts/5    00:00:02 /usr/lib/firefox/
5         firefox
4 $ kill -1 3215856
```

Observe that firefox will close, even though it was running in the background. This is because the shell will propagate the hang-up signal to the child process. If the parent shell was forcefully killed using the **SIGKILL** signal, then it won't have the opportunity to propagate the hang-up signal to the child process. This is a separate process than the natural killing of firefox running in foreground even when shell is killed with **SIGKILL** signal.

Now, to fix this, we can simply run the `disown` command in the terminal where we started the `firefox` process.

Again open a terminal emulator and run the following command.

```
1 $ firefox &
2 $ disown
3 $
```

Now, in the other terminal, run the following command.

```
1 $ ps -efj | awk "\$2==$(pgrep firefox);NR==1"
2   UID          PID      PPID      PGID      SID  C
3           STIME TTY          TIME CMD
4 sayan      3216429 3215856 3216429 3215856 69
5           18:45 pts/5    00:00:02 /usr/lib/firefox/
6           firefox
7 $ kill -1 3215856
8 $ ps -efj | awk "\$2==$(pgrep firefox);NR==1"
9   UID          PID      PPID      PGID      SID  C
10          STIME TTY          TIME CMD
11 sayan      3216429          1 3216429 3215856 10
12          18:45 ?    00:00:03 /usr/lib/firefox/
13           firefox
```

Firefox does not close anymore, even when the parent process is hanged up.

### 4.3.2 Jobs

To list the jobs that are running in a shell, you can use the `jobs` command.

```
1 $ firefox &
2 $ sleep 50 &
3 $ jobs
4 [1]-  Running                  firefox &
5 [2]+  Running                  sleep 50 &
```

Here + denotes the current job, and - denotes the previous job. The first column is the job number, it can also be used to refer to the job inside that same shell. The process ID of a process can be used to refer to the process from anywhere, but the job ID is only valid in the shell where it is created.

The process id can be listed using the `jobs -l` command.

```
1 $ jobs -l
2 [1]- 3303198 Running                 firefox &
3 [2]+ 3304382 Running                 sleep 50
   &
```

Using `disown` removes the job from this table. We can selectively remove only some jobs from the table as well.

```
1 $ jobs
2 [1]- Running                 firefox &
3 [2]+ Running                 sleep 50 &
4 disown %1
5 $ jobs
6 [2]+ Running                 sleep 50 &
```

Whereas using `disown -a` will remove all jobs from the table. `disown -r` will remove only running jobs from the table.

If you don't really want to lose the job from the table, but you want to prevent it from being killed when the shell is killed, you can use `disown -h` to mark the jobs to be ignored by the hang-up signal. It will have same effect as last exercise, but it will still be present in the output of the `jobs` command.

### 4.3.3 Suspending and Resuming Jobs

Sometimes you may want to pause a job and resume it later. This is supported directly by the linux

6: The difference between **SIGSTOP** and **SIGTSTP** is that **SIGSTOP** is a signal that cannot be caught or ignored by the process, so the process will be paused immediately. **SIGTSTP** is a signal that can be caught or ignored by the process, so the process can do some cleanup before pausing. The default action of **SIGTSTP** is to pause the process.

kernel. To pause any process you can send it the **SIGSTOP** or **SIGTSTP** signal.<sup>6</sup> This can be done using the same **kill** command. The signal number for **SIGSTOP** is 19, and for **SIGTSTP** is 20.

To resume the process, you can send it the **SIGCONT** signal. The signal number for **SIGCONT** is 18.

**Exercise 4.3.1** Try to pause a job using the **SIGSTOP** signal, then resume it using the **SIGCONT** signal. Open firefox from a terminal using `firefox &` and note the PID, then pause it using `kill -19 <PID>`, try to click on the firefox window, and see if it responds. Then resume it using `kill -18 <PID>`. Does the firefox window respond now?

If you start a command from the shell without using the `&` operator, you can pause the command using `Ctrl+Z` and resume it using the `fg` command. This sends the same signals as above, and uses the shell's job table to keep track of the jobs.

**Remark 4.3.1** Just like `disown`, the `fg` command can also take the job number as an argument to bring that job to the foreground. The default job is the current job. (Marked with a `+` in the `jobs` command)

You can also use the `bg` command to resume a job, but in the background. This has same effect as using the `&` operator at the end of the command.

**Remark 4.3.2** Since the `disown`, `fg`, and `bg` commands work on the shell's job table, they are

shell builtins, and not a executable binary. You can verify this using the type command.

You cannot perform job control on a process that is not started from the shell, or if you have disowned the process.

#### 4.3.4 Killing Processes

We have been using the `kill` command to send signals to processes. The `kill` command is a shell builtin and an executable command that is used to send signals to processes. The default signal that is sent is the **SIGTERM** signal, which is signal number 15. The `kill` command is the user-land way to communicate with the kernel that some process needs to be given a certain signal.

##### Syntax

```
1| $ kill [-signal|-s signal] PID|name...
```

Let us also briefly discuss what the synopsis of the command means, and how to interpret it.

The first word is the name of the command, which is `kill` in this case. The argument **signal** inside square brackets means that it is optional. The argument **PID** is the process ID of the process that you want to kill. The argument **name** is the name of the process that you want to kill. The pipe between **PID** and **name** means that you can provide either the PID or the name of the process. The ellipsis (...) after **PID | name** means that you can provide as many PIDs or names as you want.

**Remark 4.3.3** As mentioned, `kill` is also a shell builtin command. This means that the synopsis

seen in the man page of kill is not the same as the synopsis of the builtin. The bash builtin of kill does not support providing names of the processes, only the PIDs. Hence if you want to kill a process by its name, you will either have to use the path of the kill binary, or use the pkill command.

So for example, we can run kill in the following manners.

```
1 $ kill 2452
2 $ kill -9 2452
3 $ kill -9 2452 62
4 $ kill -SIGKILL 2525
5 $ kill -SIGKILL 2525 732
```

The kill command can also be used to send other non-terminating signals to the process. For example, the **SIGSTP** signal can be used to pause a process, and the **SIGCONT** signal can be used to resume a process. Similarly, there are some undefined signals that can be used to send custom signals to the process.

**SIGUSR1** and **SIGUSR2** are signals that do not have any predefined behaviour, and can be used by the user to send custom signals to the process. The behaviour of the process on receipt of these signals is decided by the process and told to the user by the process documentation. The user can then send these signals to the process using the kill command. This helps user interact with processes that are not running directly in the foreground of a shell.

Processes can also **trap** signals, which means that they can catch a signal and run a custom handler function. This is useful when you want to do some cleanup before the process is killed. This can also be

used to totally change how the process behaves on receipt of a signal. However, to prevent malicious code from running, the **SIGKILL** signal cannot be trapped, and the process will be killed immediately. Similarly the **SIGSTOP** signal, which is similar in definition to the **SIGSTP** signal, cannot be trapped.

To see the list of the signals, we can run `kill -l`.

```
1 $ kill -l
2   1) SIGHUP      2) SIGINT      3) SIGQUIT
3       4) SIGILL      5) SIGTRAP
4   6) SIGABRT     7) SIGBUS      8) SIGFPE
5       9) SIGKILL     10) SIGUSR1
6  11) SIGSEGV     12) SIGUSR2     13) SIGPIPE
7       14) SIGALRM     15) SIGTERM
8  16) SIGSTKFLT    17) SIGCHLD     18) SIGCONT
9       19) SIGSTOP     20) SIGTSTP
10 21) SIGTTIN     22) SIGTTOU     23) SIGURG
11       24) SIGXCPU     25) SIGXFSZ
12 26) SIGVTALRM    27) SIGPROF     28) SIGWINCH
13       29) SIGIO       30) SIGPWR
14 31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1
15       36) SIGRTMIN+2 37) SIGRTMIN+3
16 38) SIGRTMIN+4 39) SIGRTMIN+5 40) SIGRTMIN+6
17       41) SIGRTMIN+7 42) SIGRTMIN+8
18 43) SIGRTMIN+9 44) SIGRTMIN+10 45) SIGRTMIN
19       +11 46) SIGRTMIN+12 47) SIGRTMIN+13
20 48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX
21       -14 51) SIGRTMAX-13 52) SIGRTMAX-12
22 53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9
23       56) SIGRTMAX-8 57) SIGRTMAX-7
24 58) SIGRTMAX-6 59) SIGRTMAX-5 60) SIGRTMAX-4
25       61) SIGRTMAX-3 62) SIGRTMAX-2
```

Some of the important signals are:

- ▶ **SIGHUP** - Hangup signal. This is sent to a process when the terminal is closed. This is used to tell the process that the terminal is no

longer available.

- ▶ **SIGINT** - Interrupt signal. This is sent to a process when the user presses `Ctrl+C`. This is used to tell the process to stop what it is doing and exit.
- ▶ **SIGKILL** - Kill signal. This is used to kill a process immediately. This signal cannot be caught or ignored by the process.
- ▶ **SIGTERM** - Terminate signal. This is used to tell the process to exit gracefully. The process can catch this signal and do some cleanup before exiting. This is the default signal sent by the `kill` command.
- ▶ **SIGSTP** - Stop signal. This is used to pause a process. This is sent when the user presses `Ctrl+Z`. This signal can be caught or ignored by the process.
- ▶ **SIGSTOP** - Stop signal. This is used to pause a process. This signal cannot be caught or ignored by the process.
- ▶ **SIGCONT** - Continue signal. This is used to resume a process that has been paused using the **SIGSTOP** signal. This is sent when the user presses `fg` or `bg`.
- ▶ **SIGUSR1** - User defined signal 1. This is a signal that can be used by the user to send a custom signal to the process.
- ▶ **SIGUSR2** - User defined signal 2. This is a signal that can be used by the user to send a custom signal to the process.
- ▶ **SIGCHLD** - Child signal. This is sent to the parent process when a child process exits. This is used to tell the parent process that the child process has exited.
- ▶ **SIGSEGV** - Segmentation fault signal. This is sent to a process when it tries to access memory that it is not allowed to access.
- ▶ **SIGPIPE** - Pipe signal. This is sent to a process

when it tries to write to a pipe that has been closed.

## 4.4 Finding Processes

As we saw, managing a process is easy once we know the PID of the process. However, it is not always easy to find the PID of a process. To do this, there are multiple tools in linux that can be used.

### 4.4.1 pgrep

The `pgrep` command is used to find the PID of a process based on its name. It can take the name of the process as an argument, and will print the PID of the processes that match the name. The search can also be a regex pattern.<sup>7</sup> We have already seen how `pgrep` can be used to find the PID of a process.

<sup>7</sup>: We will discuss regex patterns in the next chapter.

```
1 | $ pgrep firefox
2 | 526272
```

We can also use it for any process run from the terminal.

```
1 | $ sleep 50 &
2 | $ sleep 20 &
3 | $ sleep 10 &
4 | $ pgrep sleep
5 | 98963
6 | 99332
7 | 99526
```

### 4.4.2 pkill

Similarly, we also have the `pkill` command, which is used to kill a process based on its name. It can take the name of the process as an argument, and will send the **SIGTERM** signal to the processes that match the name. Other signals can also be sent using the `-signal` flag, similar to the `kill` command.

```
1| $ pkill firefox
```

### 4.4.3 pidwait

The `pidwait` command is used to wait for a process to exit. It searches for the process using its name, a part of its name, or any regex matching its name, and waits for the process to exit.

**Exercise 4.4.1** Open firefox from a terminal in the background, then use the `pidwait` to wait till the process exits. After some time, close the firefox window and observe the terminal.

## 4.5 Listing Processes

Sometimes we may not even know the name of the process, and we may want to list all the processes running on the system. This can be done using the many commands.

### 4.5.1 ps

8: It exists in the Unix V7 manual, which was released in 1979. It has BSD-like options, GNU-like options, and System V-like options.

**ps** is an ancient command <sup>8</sup> that is used to list the processes running on the system.

There are a lot of options and flags that can be used with the **ps** command. The flags are of multiple types, and some flags perform the same function but are named differently. This is because the **ps** command has been around for a long time, and has been implemented in multiple ways in different systems. There are also different formats in which the output can be displayed.

The most common flags used with the **ps** command are:

- ▶ **ps** - This will get a snapshot of the processes owned by the user tied to the TTY.
- ▶ **ps -e** - This will show all the processes.
- ▶ **ps -f** - This will show full format listing.
- ▶ **ps -l** - This will show long format listing.
- ▶ **ps u** - This will show user-oriented format listing.
- ▶ **ps x** - This will show processes without controlling terminals.
- ▶ **ps -A** - This will show all processes.
- ▶ **ps aux** - This is a common command to see all processes owned by all users with and without TTY associations and showing the user who owns them.

- ▶ **ps -forest** - This will show the processes in a tree form.

There are hundreds of flags that can be used with the **ps** command.

**Exercise 4.5.1** Try to use the **ps** command with the flags mentioned above, and see the output of the command.

## 4.5.2 pstree

The **pstree** command is used to display the processes in a tree form. Although the **ps** command can also display the processes in a tree form using the **--forest** flag, the **pstree** command is more suited for this purpose. It has many features that **ps** lacks, such as collapsing branches of identical processes, better ASCII art, Unicode support, etc.

If the system and the terminal supports unicode, the **pstree** command will automatically use **VT100 box-drawing characters** to make the tree look better. We can still force it to use ASCII with the **-A** flag.

We can also disable the clubbing of identical processes using the **-c** flag.

The **pstree** command optionally takes a PID as an argument, and will display the tree rooted at that PID. If no PID is provided, it will display the tree rooted at the **init** process.

I can find the PID of the **tmux** server I am running to develop this book using the **pgrep** command.

```
1 | $ pgrep tmux
2 | 62957
```

And then use the `pstree` command to display the tree rooted at that PID.

```

1 $ pstree -A 62957
2 tmux: server---bash---nvim---nvim---node
3 |           |           |
3 ---14*[{node}]
4 |           |           |           |-texlab
5 |           |           |           |
5 ---9*[{texlab}]
6 |           |           |           |
6 ---9*[{nvim}]
7 |           |           |           '-2*[{nvim}]
8 |           |           |           '-bash---watch.sh---entr
9 |           |           |           '-bash---zathura---8*[{zathura}]
10 |           |           |           '-3*[bash]
11 |           |           |           '-2*[bash---man---less]
11 |           |           |           '-bash---nvim---nvim---2*[node
11 ---9*[{node}]
12 |           |           |           '-{nvim}
13 |           |           |           '-2*[{nvim}]
14 |           |           '-bash---pstree
15 |           |           '-xsel

```

This helps us easily find out which processes are running under which process, and helps us understand the process tree.

### 4.5.3 top

The `top` command is used to display the processes that are running in real time. It is an interactive command that displays the processes in a table format, and updates the table every few seconds. It also displays the CPU and memory usage of the processes.

The `top` command is very useful when you want to monitor the processes and the resources they are using in real time. It is also useful when you want to find out which process is using the most CPU or memory.

Since it is an interactive command, it has keyboard shortcuts as well, along with runtime options that can be used to change the behaviour of the command.

```

1 $ top
2 top - 15:44:55 up 40 min,  1 user,  load
      average: 2.03, 1.37, 1.02
3 Tasks: 271 total,   1 running, 270 sleeping,
      0 stopped,   0 zombie
4 %Cpu(s):  9.8 us,  4.9 sy,  0.0 ni, 85.4 id,
      0.0 wa,  0.0 hi,  0.0 si,  0.0 st
5 MiB Mem : 7764.2 total,   604.5 free,
      5663.2 used,   1919.6 buff/cache
6 MiB Swap: 20002.0 total, 19901.2 free,
      100.8 used.   2101.0 avail Mem
7
8      PID USER      PR  NI    VIRT    RES    SHR
9          S %CPU  %MEM      TIME+ COMMAND
10     631 sayan     20   0 1248084  81976  44580
11          S 18.2   1.0  1:39.82 Xorg
10    1072 sayan     20   0 3159356 229336  89816
11          S  9.1   2.9  0:44.01 spotify
11    1079 sayan     20   0 1123.5g 128252  65776
12          S  9.1   1.6  0:09.37 Discord
12    1 root       20   0   22076  12840   9476
13          S  0.0   0.2  0:03.62 systemd
13    2 root       20   0       0       0       0
14          S  0.0   0.0  0:00.00 kthreadd
14    3 root       20   0       0       0       0
14          S  0.0   0.0  0:00.00 pool_wo+

```

Note: The output of the **top** command is not static, and it contains control characters and [ANSI escape codes](#) to make the output beautiful and interactive. This is why the output is not suitable for use in scripts, and is only meant for human consumption. I have removed the control characters and ANSI escape codes from the output shown here.

#### 4.5.4 htop

The **htop** command is an interactive process viewer for Unix systems. It is inspired from the **top** command, but has a lot more features such as scrolling the process list, searching for processes, killing processes, tree view of processes, etc.

**Exercise 4.5.2** Run (Install if not present) the **htop** command and see the output. Notice how the output is more interactive and colourful than the **top** command. Run something heavy in the background, and see how the CPU and the memory usage changes in real time.

One such command to run to simulate heavy CPU usage is to run the following command.

```
1 | $ cat /dev/urandom | gzip > /dev/null &
```

This will compress the random data from `/dev/urandom` and write it to `/dev/null`. This will use a lot of CPU, and you can see the CPU usage spike up. But this is a single core process, so the CPU usage will be limited to only one core. However, the CPU core used may keep changing.

**Remark 4.5.1** The above command is a simple way to generate CPU usage. It will not write any data to disk and not eat any disk space. However, the command should be typed carefully, since if you forget to add the `> /dev/null` part, it will write the compressed data to the terminal if `-f` is given to `gzip`, and the terminal will be filled with random data and get messed up. In this case you can type `tput reset` to reset the terminal after killing the process.

## 4.5.5 btop

The **btop** command is a terminal based graphical process viewer. It is inspired from the **htop** command, but has a more graphical interface. It is

written in python, and uses the **blessings** library to draw the interface.

#### 4.5.6 **glances**

The **glances** command is a cross-platform monitoring tool that is used to monitor the system resources in real time. It is written in python, and uses the **curses** library to draw the interface.

## 4.6 Exit Codes

Every process that runs in linux has an exit code when it terminates. This is a number that is returned by the process to the parent process when it exits. This number is used to tell the parent process if the process exited successfully or not. The exit code is a number between 0 and 255, and is used to tell the parent process the status of the child process.

The exit code of the last run process is stored in a variable called `$?` in the shell. Successful processes return 0, whereas unsuccessful processes return a non-zero number. The exact return code is decided by the process itself, and usually has some meaning to the process. Some common exit codes are:

- ▶ 0 - Success
- ▶ 1 - General error
- ▶ 2 - Misuse of shell builtins
- ▶ 126 - Command invoked cannot execute
- ▶ 127 - Command not found
- ▶ 128 - Invalid argument to exit
- ▶ 128+n - Fatal error signal "n"
- ▶ 130 - Script terminated by `Ctrl+C`
- ▶ 137 - Process killed with **SIGKILL**
- ▶ 255 - Exit status out of range

**Remark 4.6.1** Note that  $130 = 128 + 2$ , and 2 is the signal number for **SIGINT**, thus, the exit code for a process that is terminated by `Ctrl+C` is 130. Similarly, any other signal sent to the process causing it to exit abnormally will have an exit code of  $128 + n$ , where  $n$  is the signal number. Similarly  $137 = 128 + 9$ , and 9 is the signal number for **SIGKILL**.

To return any exit status from your script, you can use the `exit` command.

```
1 $ cat myscript.sh
2 #!/bin/bash
3 echo "hello"
4 exit 25
5 $ ./script.sh
6 bash: ./script.sh: No such file or directory
7 $ echo $?
8 127
9 $ ./myscript.sh
10 bash: ./myscript.sh: Permission denied
11 $ echo $?
12 126
13 $ chmod u+x myscript.sh
14 $ ./myscript.sh
15 hello
16 $ echo $?
17 25
```

The exit code is how shell constructs like `if`, `while`, and `until` construct their conditions. If the exit code is 0, then it is considered true, and if the exit code is non-zero, then it is considered false.

```
1 $ if ./myscript.sh; then echo "success"; else
2     echo "failure $?"; fi
3 hello
4 failure 25
5 $ if ls /bin/bash; then echo "success"; else
6     echo "failure $?"; fi
7 /bin/bash
8 success
```



# Streams, Redirections, Piping

5

## 5.1 Multiple Commands in a Single Line

Sometimes we may want to run multiple commands in a single line. For example, we may want to run two commands `ls` and `wc` in a single line. We can do this by separating the commands with a semicolon. This helps us see the output of both the commands without having the prompt in between.

For example, the following command will run `ls` and `wc` in a single line.

```
1 $ ls; wc .bashrc
2 docs      down    music   pics   programs   scripts
            tmp     vids
3 340 1255 11238 .bashrc
```

In this way of executing commands, the success or failure of one command does not affect the other command. Concisely, the commands are executed independently and sequentially. Even if a command fails, the next command will be executed.

```
1 $ date; ls /nonexistent ; wc .bashrc
2 Wed Jul  3 06:54:45 PM IST 2024
3 ls: cannot access '/nonexistent': No such file
       or directory
4 340 1255 11238 .bashrc
```

### 5.1.1 Conjunction and Disjunction

We can also run multiple commands in a single line using conjunction and disjunction. The conjunction

operator `&&` is used to run the second command only if the first command is successful. The disjunction operator `||` is used to run the second command only if the first command fails. In computer science, these operators are also known as short-circuit logical **AND** and **OR** operators.<sup>1</sup>

- 1: A short-circuit logical **AND** operator returns **true** if both the operands are **true**.  
 If the first operand is **false**, it does not evaluate the second operand and returns **false** directly.

```

1 $ ls /nonexistant && echo "ls successful"
2 ls: cannot access '/nonexistant': No such file
   or directory
3 $ ls /home && echo "ls successful"
4 lost+found sayan test1
5 ls successful

```

In the first command, the `ls` command fails, so the `echo` command is not executed. In the second command, the `ls` command is successful, so the `echo` command is executed.

The success or failure of a command is determined by the exit status of the command. If the exit status is 0, the command is successful. If the exit status is non-zero, the command is considered to have failed.

The exit status of the last command can be accessed using the special variable `$?`. This variable contains the exit status of the last command executed.

```

1 $ ls /nonexistant
2 ls: cannot access '/nonexistant': No such file
   or directory
3 $ echo $?
4 2

```

Here, the exit status of the `ls` command is 2, because the file `/nonexistant` does not exist.

```

1 $ ls /home
2 lost+found sayan test1
3 $ echo $?
4 0

```

Here, the exit status of the `ls` command is 0, because the directory `/home` exists, and the command is successful.

Similarly, we can use the disjunction operator `||` to run the second command only if the first command fails.

```
1 $ ls /nonexistant || echo "ls failed"
2 ls: cannot access '/nonexistant': No such file
   or directory
3 ls failed
```

In this case, the `ls` command fails, so the `echo` command is executed. However, since the disjunction operator is a short-circuit operator, the `echo` command is executed only if the `ls` command fails. If the `ls` command is successful, the `echo` command is not executed.

```
1 $ ls /home || echo "ls failed"
2 lost+found sayan test1
```

In this case, the `ls` command is successful, so the `echo` command is not executed.

We can also chain multiple commands using conjunction and disjunction.

```
1
2 $ date && ls /hello || echo "echo"
3 Thu Jul  4 06:53:08 AM IST 2024
4 ls: cannot access '/hello': No such file or
   directory
5 echo
```

In this case, the `date` command is successful, so the `ls` is executed. The `ls` command fails, so the `echo` command is executed.

However, even if the first command fails, and the second command is skipped, the third command is executed.

```

1 $ ls /hello && date || echo echo
2 ls: cannot access '/hello': No such file or
   directory
3 echo

```

In this case, the `ls` command fails, so the `date` command is not executed. However, the `echo` command is executed as the exit status of the `ls` command is non-zero.

To make the `echo` command execute only if the `date` command is run and fails, we can use parentheses.

```

1 $ ls /hello && (date || echo echo)
2 ls: cannot access '/hello': No such file or
   directory

```

Although the parentheses look like grouping a mathematical expression, they are actually a subshell. The commands inside the parentheses are executed in a subshell. The exit status of the subshell is the exit status of the last command executed in the subshell.

We can see the subshell count using the `echo \$BASH\_SUBSHELL`.

```

1 $ echo $BASH_SUBSHELL
2 0
3 $ (echo $BASH_SUBSHELL)
4 1
5 $ (:;(echo $BASH_SUBSHELL ))
6 2
7 $ (:;(:;(echo $BASH_SUBSHELL )))
8 3

```

**Remark 5.1.1** When nesting in more than one subshell, simply putting two parentheses side by side will not work. This is because the shell will

interpret that as the mathematical evaluation of the expression inside the parentheses. To avoid this, we can use a colon : no-op command followed by a semicolon ; to separate the two parentheses.

**Remark 5.1.2** Setting up an environment takes up time and resources. Thus, it is better to avoid creating subshells unless necessary.

## 5.2 Streams

There are three standard streams in Unix-like operating systems:

1. **Standard Input (stdin)**: This is the stream where the input is read from. By default, the standard input is the keyboard.
2. **Standard Output (stdout)**: This is the stream where the output is written to. By default, the standard output is the terminal.
3. **Standard Error (stderr)**: This is the stream where the error messages are written to. By default, the standard error is the terminal.

There are also other numbered streams, such as 3, 4, etc., which can be used to read from or write to files.

However, sometimes a process may need to take input from a file or send output to a file. When this is required, the standard stream is mapped to a file. This is known as **redirection**.

To maintain which file or resource is mapped to which stream, the operating system maintains a table known as the **file descriptor table**. The file descriptor table is a table that maps the file descriptors to the files or resources.

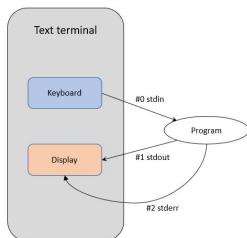


Figure 5.1: Standard Streams

**Definition 5.2.1 (File Descriptor)** A **file descriptor** is a process-unique identifier that the operating system assigns to a file or resource. The file descriptor is an integer that is used to identify the file or resource.

The default file descriptors are:

1. **Standard Input (stdin)**: File descriptor 0

2. **Standard Output (stdout):** File descriptor 1
3. **Standard Error (stderr):** File descriptor 2

In the traditional implementation of Unix, file descriptors index into a per-process file descriptor table maintained by the kernel, that in turn indexes into a system-wide table of files opened by all processes, called the file table. This table records the mode with which the file (or other resource) has been opened: for reading, writing, appending, and possibly other modes. It also indexes into a third table called the inode table<sup>2</sup> that describes the actual underlying files. To perform input or output, the process passes the file descriptor to the kernel through a system call, and the kernel will access the file on behalf of the process. The process does not have direct access to the file or inode tables.

The file descriptors of a process can be inspected in the `/proc` directory. The `/proc` directory is a pseudo-filesystem that provides an interface to kernel data structures. The `/proc/PID/fd` directory contains the file descriptors of the process with the PID.

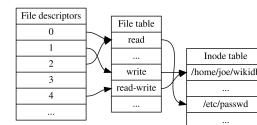
Most GNU core utilities that accept a file as an argument will also work without the argument and will read from the standard input. This behaviour lets us chain commands together using pipes easily without any explicit file handling.

```

1 $ cat
2 hello
3 hello
4 This command is repeating the input
5 This command is repeating the input
6 Press Ctrl+D to exit
7 Press Ctrl+D to exit

```

2: We have covered inodes and inode tables in detail in the Chapter 1 chapter.



**Figure 5.2:** File Descriptor Table

The `cat` command is usually used to print the contents of one or more files. However, when no file

is provided, it reads from the standard input. In this case, the standard input is the keyboard. This is very useful when we want to repeat the input or when we want to read from the standard input.

**Question 5.2.1** Can we also use cat to write to a file?

**Answer 5.2.1** Yes, we can use the cat command to write to a file. When no file is provided, the cat command reads from the standard input. So the input will be printed back to the standard output. However, if we can somehow change the standard output to a file, then the input will be written to the file.

## 5.3 Redirection

Redirection is the process of changing the standard streams of a process. This is done by the shell before the process is executed. The shell uses the `<`, `>`, and `2>` operators to redirect the standard input, standard output, and standard error streams of a process.

As the shell is responsible for redirection, the process is not aware of the redirection. The process reads from or writes to the file descriptor it is given by the shell. The process does not know whether the file descriptor is a file, a terminal, or a pipe.

However, there are ways for a process to guess whether the file descriptor is a terminal or a file. This is done by using the `isatty()` function. The `isatty()` function returns `1` if the file descriptor is a terminal, and `0` if the file descriptor is a file.

### 5.3.1 Standard Output Redirection

The standard output of a process can be redirected to a file using the `>` operator. Observe the following example.

```
1 $ date > date.txt
2 $ cat date.txt
3 Thu Jul  4 07:37:27 AM IST 2024
```

Here, the output of the `date` command is redirected to the file `date.txt`. The `cat` command is then used to print the contents of the file `date.txt`.

The process did not print the output to the terminal. Instead, the shell changed the standard output of the process to the file `date.txt`.

We can also use redirection to create a file. If the file does not exist, the shell will create the file. If the file exists, the shell will truncate the file.

```
1 $ > empty.txt
```

This command creates an empty file `empty.txt`. The `>` operator is used to redirect the output of the command to the file `empty.txt`. Since there is no output, the file is empty.

We can also use redirection along with the `echo` command to create a file with some content.

```
1 $ echo "hello" > hello.txt
2 $ cat hello.txt
3 hello
```

Here, the output of the `echo` command is redirected to the file `hello.txt`. The `cat` command is then used to print the contents of the file `hello.txt`.

Recall that the `cat` command will read from the standard input if no file is provided. We can use

this to create a file with the input from the standard input.

```

1 $ cat > file.txt
2 hello, this is input typed from the keyboard
3 I am typing more input
4 all this is being written to the file
5 to stop, press Ctrl+D
6 $ cat file.txt
7 hello, this is input typed from the keyboard
8 I am typing more input
9 all this is being written to the file
10 to stop, press Ctrl+D

```

It is important to note that the process of creating the file if it does not exist, and truncating the file if it exists, along with redirecting the output, is done by the shell, not the process. All of this is done before the process is executed. This creates an interesting exemplar when using redirection with the `ls` command.

### Output contains output

```

1 $ ls
2 hello
3 $ ls > output.txt
4 $ cat output.txt
5 hello
6 output.txt

```

Since the file is created even before the process is executed, the file is also a part of the output of the `ls` command. This is why the file `output.txt` is also printed by the `cat` command.

### Output format changes

Another interesting example using the `ls` command is when the directory contains more than one file. The output of `ls` is usually formatted to fit the terminal. However, when the output is redirected

to a file, the output is not in a column format. Instead, each file is printed on a new line.

```
1 $ ls
2 hello  output.txt
3 $ ls > output.txt
4 $ cat output.txt
5 hello
6 output.txt
```

Observe that the output of the `ls` command is not in a column when redirected to a file. But how does `ls` know that the output is not a terminal? It uses the file descriptor to guess whether the output is a terminal or a file. If the file descriptor is a terminal, then the output is formatted to fit the terminal. If the file descriptor is a file, then the output is not formatted.

However, we can always force the output to be single-column using the `-1` option.

```
1 $ ls -1
2 hello
3 output.txt
```

This behaviour is not limited to the `ls` command. Most commands usually strip out the formatting when the output is redirected to a file.

If you are using a terminal that supports ANSI escape codes<sup>3</sup>, you can use the `ls` command with the `--color=auto` option to get colored output. However, when the output is redirected to a file, the ANSI escape codes are not printed. This is because the ANSI escape codes are not printable characters. They are control characters that tell the terminal to change the color of the text.

3: ANSI escape codes are special sequences of characters that are used to control the terminal. For example, to change the color of the text, the ANSI escape code `[31m` is used. Similarly other ANSI escape codes are used to change the text style, and the position and state of the cursor.

## Output and Input from same file

If you try to redirect the output of a command to a file, and then also try to read from the same file, you will get an empty file. This is because the file is truncated before the command is executed.

```

1 $ echo "hello" > output.txt
2 $ cat output.txt
3 hello
4 $ cat output.txt > output.txt
5 $ cat output.txt
6 $
```

**Remark 5.3.1** Although we can simply use `>` to redirect the output to a file, the full syntax is `1>`. This is because the file descriptor for the standard output is 1. The `1>` is used to redirect the standard output to a file. However, since the standard output is the default output, we can omit the 1 and use only `>`.

### 5.3.2 Standard Error Redirection

The standard error of a process can be redirected to a file using the `2>` operator. Observe the following example.

```

1 $ ls /nonexistant 2> error.txt
2 $ cat error.txt
3 ls: cannot access '/nonexistant': No such file
   or directory
```

Here, the error message of the `ls` command is redirected to the file `error.txt`. The `cat` command is then used to print the contents of the file `error.txt`.

It is important to realise that each process has two streams to output to, the standard output and the

standard error. The standard output is usually used to print the output of the process, while the standard error is used to print the error messages.

This helps us differentiate between the output and the error messages. Also, if the output of a process is redirected to a file, the error will still be printed to the terminal. This is because the standard error is not redirected to the file. This makes debugging easier, as the error messages are not lost.

```

1 $ ls -d /home /nonexistant > output.txt
2 ls: cannot access '/nonexistant': No such file
      or directory
3 $ cat output.txt
4 /home

```

Here, the output of the `ls` command is redirected to the file `output.txt`. However, the error message is still printed to the terminal.

We can redirect both the standard output and the standard error to files using the `>` and `2>` operators.

```

1 $ ls -d /home /nonexistant > output.txt 2>
      error.txt
2 $ cat output.txt
3 /home
4 $ cat error.txt
5 ls: cannot access '/nonexistant': No such file
      or directory

```

### Redirecting both streams to the same file

Lets try to redirect both the standard output and the standard error to the same file.

```

1 $ ls -d /home /nonexistant > output.txt 2>
      output.txt
2 $ cat output.txt
3 /home

```

```
4| nnot access '/nonexistant': No such file or
   directory
```

- 4: The output message is /**home**, followed by a newline character. This makes the output message 6 characters long. The shell first writes the error message to the file (because that is printed first by the ls command), and then overwrites the first 6 bytes of the file with the output message. Since the 6th byte is a newline character, it looks like there are two lines in the file.

Why did the error message get mangled? This is because the shell truncates the file before the process is executed. So the error is written to the file, and then the output message is written to the same file, overwriting the error partially. Observe that only the first six characters of the error message are mangled, the same size of the output.<sup>4</sup>

The correct way to redirect both the standard output and the standard error to the same file is to use the 2>\&1 operator. This means that the standard error is redirected to the standard output. Here, the 1 is the file descriptor for the standard output. The & is used to tell the shell that the 1 is a file descriptor, not a file.

```
1| $ ls -d /home /nonexistant > output.txt 2>&1
2| $ cat output.txt
3| ls: cannot access '/nonexistant': No such file
   or directory
4| /home
```

However, the order is important. The 2>\&1 operator should be placed at the end of the command. If it is placed at the beginning, then the standard error will be redirected to the standard output, which, at that point, is the terminal. Then the standard output will be redirected to the file. Thus, only the standard output will be redirected to the file.

```
1| $ ls -d /home /nonexistant 2>&1 > output.txt
2| ls: cannot access '/nonexistant': No such file
   or directory
3| $ cat output.txt
4| /home
```

### 5.3.3 Appending to a File

The > operator is used to redirect the output to a file. If the file does not exist, the shell will create the file. If the file exists, the shell will truncate the file. However, if we want to append the output to the file, we can use the >> operator.

```

1 $ echo "hello" > hello.txt
2 $ cat hello.txt
3 hello
4 $ echo "world" > hello.txt
5 $ cat hello.txt
6 world
7 $ echo "hello" > hello.txt
8 $ echo "world" >> hello.txt
9 $ cat hello.txt
10 hello
11 world

```

Observe that the > operator truncates the file, while the >> operator appends to the file.

We can also append the standard error to a file using the 2>>

```

1 $ ls /nonexistent 2> error.txt
2 $ cat error.txt
3 ls: cannot access '/nonexistent': No such file
      or directory
4 $ daet 2>> error.txt
5 $ cat error.txt
6 ls: cannot access '/nonexistent': No such file
      or directory
7 bash: daet: command not found

```

This is useful when we want to append the error messages to a file, like in a log file.

### Circular Redirection

If we try to redirect the output of a command to the same file that we are reading from, we will get an

empty file. However, if we append the output to the file, we will get an infinite loop of the output. This should not be done, as it will fill up the disk space. However, GNU core utilities `cat` is smart enough to detect this and will not read from the file.

```

1 $ echo hello > hello
2 $ cat hello >> hello
3 cat: hello: input file is output file
4 $ cat < hello > hello
5 cat: -: input file is output file

```

5: We will cover pipes in the next section.

However, it can still be tricked by using a pipe.<sup>5</sup>

```

1 $ echo hello > hello
2 $ cat hello | cat >> hello
3 $ cat hello
4 hello
5 hello

```

Here we dont get into an infinite loop because the first `cat` command reads from the file and writes to the pipe. The second `cat` command reads from the pipe and writes to the file. Since the file is not read from and written to at the same time, we dont get into an infinite loop.

However, BSD utilities like `cat` do not have this check, thus giving the same input and output file will result in an infinite loop.

### 5.3.4 Standard Input Redirection

The standard input of a process can be redirected from a file using the `<` operator. Although most commands accept a filename in the argument to read from, so we can directly provide the file in the argument. However, some commands do not accept a filename and only read from the standard

input. In such cases, we can use the < operator to redirect the standard input from a file.

```
1 $ wc < ~/.bashrc
2   340  1255 11238
```

Although the `wc` command accepts a filename as an argument, we can also redirect the standard input from a file using the < to the command. However, now the output of the `wc` command is different than when we provide the filename as an argument.

```
1 $ wc ~/ .bashrc
2   340  1255 11238 /home/sayan/.bashrc
```

When we provide the filename as an argument, the `wc` command prints the number of lines, words, and characters in the file, followed by the filename. However, when we redirect the standard input from a file, the `wc` command prints only the number of lines, words, and characters in the file. This is because it does not know that the input is coming from a file. For `wc`, the input is just a stream coming from the standard input.

```
1 $ wc
2 Hello, this is input from the keyboard
3 I can simply type more input here
4 All of this is taken as standard input
5 by wc command
6 Stop by pressing Ctrl+D
7      5       29      150
```

Another example where giving filename is not possible is the `read` command. The `read` command reads a line from the standard input and assigns it to a variable. The `read` command does not accept a filename as an argument. So we have to use the < operator to redirect the standard input from a file.

```
1 $ read myline < /etc/hostname
```

```

2 $ echo $myline
3 rex

```

Here, the read command reads a line from the file /etc/hostname and assigns it to the variable myline. The echo command is then used to print the value of the variable myline. To access the value of the variable, we use the \$ operator.<sup>6</sup>

6: This will be covered in depth in the Chapter 8 chapter.

### 5.3.5 Here Documents

A **here document** is a way to provide input to a command without using a file. The input is provided directly in the command itself. This is done by using the << operator followed by a delimiter.

```

1 $ wc <<EOF
2 This is acting like a file
3 however the input is directly provided
4 to the shell or the script
5 this is mostly used in scripts
6 we can use any delimiter instead of EOF
7 to stop, type the delimiter on a new line
8 EOF
9          6        41      206

```

We can optionally provide a hyphen - after the << to ignore leading tabs. This is useful when the here document is indented.

```

1 $ cat <<EOF
2 > hello
3 >      this is space
4 >      this is tab
5 > EOF
6 hello
7      this is space
8          this is tab
9 $ cat <<-EOF
10 hello

```

```

11      this is space
12          this is tab
13 EOF
14 hello
15      this is space
16 this is tab

```

The delimiter can be any string. However, it is common to use EOF as the delimiter. The delimiter should be on a new line.

```

1 $ wc <<END
2 If I want to have
3 EOF
4 as part of the input
5 then I can simply change the delimiter
6 END
7      4      18      82

```

Here-documents also support variable expansion. The shell will expand the variables before passing the input to the command. Thus it is really useful in scripts if we want to provide a template which is filled with the values of the variables.

```

1 $ name="Sayam"
2 $ cat <<EOF
3 Hello, $name
4 This is a here document
5 EOF
6 Hello, Sayan
7 This is a here document

```

### 5.3.6 Here Strings

A **here string** is a way to provide input to a command without using a file. The input is provided directly in the command itself. This is done by using the <<< operator followed by the input. It is very similar to the here document, but the input is

provided in a single line. Due to this, a delimiter is not required.

```
1 | $ wc <<< "This is a here string"
2 |           1      5     22
```

It can also perform variable expansion.

```
1 | $ cat <<< "Hello, $USER"
2 | Hello, sayan
```

**Remark 5.3.2** Both heredocs and herestrings are simply syntactic sugar. They are similar to using echo and piping the output to a command. However, it requires one less process to be executed.

## 5.4 Pipes

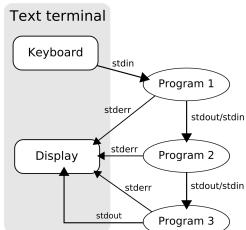


Figure 5.3: Pipes

Pipes are the holy grail of Unix-like operating systems. They are the most important concept to understand in Unix-like operating systems.

**Definition 5.4.1 (Pipe)** A **pipe** is a way to connect the standard output of one process to the standard input of another process. This is done by using the `|` operator.

Think of the shell as a factory, and the commands as machines in the factory. Pipes are like conveyor belts that connect the machines. It would be a pain if we had to manually collect the produce of one machine and then feed it to the next machine. Conveyors make it easy by automatically taking the output of one machine and feeding it to the next machine.

Pipes are the same, but for processes in shell.

```
1 $ date  
2 Thu Jul  4 09:05:30 AM IST 2024  
3 $ date | wc  
4      1      7      32
```

### 5.4.1 UNIX Philosophy

Each process simply takes the standard input and writes to the standard output. How those streams are connected is not the concern of the process. This way of simply doing one thing and doing it well is the Unix philosophy<sup>7</sup> which says that each program should do one thing and do it well. Each process should take input from the standard input and write to the standard output. The output of the process should be easy to parse by another process. The output of the process should not contain unnecessary information. This makes it easy to chain commands together using pipes.

<sup>7</sup>: The Unix philosophy, originated by Ken Thompson, is a set of cultural norms and philosophical approaches to minimalist, modular software development. It is based on the experience of leading developers of the Unix operating system. Read more [online](#).

### 5.4.2 Multiple Pipes

There is no limit to the number of pipes that can be chained together. It simply means that the output of one process is fed to the input of the next process. This simple but powerful construct lets the user do any and all kinds of data processing.

Imagine you have a file which contains a lot of words, you want to find which word is present the most number of times. You can either write a program in C or Python, etc., or you can use the power of pipes to do it in a single line using simple GNU coreutils.

I have a file `alice_in_wonderland.txt` which contains the entire text of the book Alice in Wonderland.

I want to find the words that are present the most number of times.

There are some basic preprocessing you would do even if you were writing a program. You would convert all the words to lowercase, and remove any punctuation. This can be done using the `tr` command. Then you would split the text into each word. This can be done using the `tr` command. Then you would find the count of each word. There are two ways of doing this, either you can use a dictionary<sup>8</sup> to store the frequency of each word, and increase it as you iterate over the entire text once. Or you can first sort the words, then simply count the number of times each word is repeated. Since repeated words would always be consecutive, you can simply count the number of repeated words without having to store the frequency of each word. This can be done using the `sort` and `uniq` commands. Finally, you would sort the words based on the frequency and print the top 10 words. This can be done using the `sort` and `head` commands.

Lets see how we actually do this using pipes.

```

1 $ ls alice_in_wonderland.txt
2 alice_in_wonderland.txt
3 $ tr 'A-Z' 'a-z' < alice_in_wonderland.txt | 
   tr -cd 'a-z' | tr ' ' '\n' | grep . | 
   sort | uniq -c | sort -nr | head
4   1442 the
5   713 and
6   647 to
7   558 a
8   472 she
9   463 it
10  455 of
11  435 said
12  357 i
13  348 alice

```

Let's go over each command one by one and see what it does.

**tr 'A-Z' 'a-z'**

`tr` is a command that translates characters. The first argument is the set of characters to be translated, and the second argument is the set of characters to translate to. Here, we are translating all uppercase letters to lowercase. This command converts all uppercase letters to lowercase. This is done to make the words case-insensitive.

```
1 $ tr 'A-Z' 'a-z' < alice_in_wonderland.txt  |
2   head -n20
3 alice's adventures in wonderland
4
5           alice's adventures in
6   wonderland
7
8           lewis carroll
9
10          the millennium fulcrum edition
11
12          3.0
13
14          chapter i
15
16          down the rabbit-hole
17
18
19          alice was beginning to get very tired of
20            sitting by her sister
21 on the bank, and of having nothing to do:
22   once or twice she had
23 peeped into the book her sister was reading,
24   but it had no
25 pictures or conversations in it, 'and what is
26   the use of a book,'
```

**Remark 5.4.1** Since the text is really big, I am going to filter all the intermediate outputs also through the head command.

**tr -cd 'a-z'**

The -c option is used to complement the set of characters. The -d option is used to delete the characters. Here, we are telling the tr command to delete all characters except lowercase letters and spaces. This is done to remove all punctuation and special characters. Observe the difference in output now.

```
1 $ tr 'A-Z' 'a-z' < alice_in_wonderland.txt |  
  tr -cd 'a-z' | head -c500  
2 alices adventures in wonderland  
    alices adventures in wonderland  
        lewis carroll  
            the millennium fulcrum edition  
                chapter i  
                    down the rabbit hole alice was  
beginning to get very tired of sitting by  
her sister on the bank and of having  
nothing to do once or twice she had peeped  
into the book her sister was reading but  
it had no pictures or conversations in it  
and what is the use of a book thought alice  
    w
```

**Remark 5.4.2** After we have removed all the punctuation, the text is now only a single line. This is because we have removed all the characters, including the newline characters. Thus to restrict the output, I am using the head -c500 instead of head -n20.

**tr '' '\n'**

The tr command is used to translate characters.

Here, we are translating spaces to newline characters. This is done to split the text into words. We are doing this so that each word is on a new line. This is helpful as the sort and uniq commands work on lines.

```

1 $ tr 'A-Z' 'a-z' < alice_in_wonderland.txt  |
   tr -cd 'a-z' | tr ' ' '\n' | head -20
2 alices
3 adventures
4 in
5 wonderland
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21 alices

```

**grep .**

Now we have each word on a new line. But observe that there are many empty lines. This is because of the multiple spaces between words and spaces around punctuation. We can remove these empty lines using the grep . command.<sup>9</sup>

```

1 $ tr 'A-Z' 'a-z' < alice_in_wonderland.txt  |
   tr -cd 'a-z' | tr ' ' '\n' | grep . | head -20
2 alices
3 adventures

```

9: We will learn more about regular expressions in the Chapter 6 chapter.

```

4 in
5 wonderland
6 alices
7 adventures
8 in
9 wonderland
10 lewis
11 carroll
12 the
13 millennium
14 fulcrum
15 edition
16 chapter
17 i
18 down
19 the
20 rabbithole
21 alice

```

Now we are almost there. We have each word on a new line. Now we can pass this entire stream of words to the `sort` command, this will sort the words.

### `sort`

Sort by default sorts the words in lexicographical order. This is useful as repeated words would be consecutive. This is important as the `uniq` command only works on consecutive lines only.

```

1 $ tr 'A-Z' 'a-z' < alice_in_wonderland.txt | 
    tr -cd 'a-z' | tr ' ' '\n' | grep . | 
    sort | head -20
2 a
3 a
4 a
5 a
6 a
7 a
8 a
9 a

```

```
10 a
11 a
12 a
13 a
14 a
15 a
16 a
17 a
18 a
19 a
20 a
21 a
```

If you run the command yourself without the head command, you can see that the words are sorted. The repeated words are consecutive. In the above output we can see that the word a is repeated many times. Now we can use the `uniq` command to count the number of times each word is repeated.

### `uniq -c`

The `uniq` command is used to remove consecutive duplicate lines. However, it also can count the number of times each line is repeated using the `-c` option.

```
1 $ tr 'A-Z' 'a-z' < alice_in_wonderland.txt | 
  tr -cd 'a-z' | tr ' ' '\n' | grep . | 
  sort | uniq -c | head -20
2      558 a
3          1 abarrowful
4          1 abat
5          1 abidefigures
6          1 able
7          79 about
8          1 aboutamong
9          1 aboutand
10         1 aboutby
11         3 abouther
12         3 aboutit
13         1 aboutthis
```

```

14      1 abouttrying
15      2 above
16      1 abranch
17      1 absenceand
18      2 absurd
19      1 acceptance
20      2 accident
21      1 accidentally

```

Great! Now we have the count of each word. However, the words are still sorted by the word. We want to sort the words by the count of the word. This can be done using the `sort` command.

### **sort -nr**

Sort by default sorts in lexicographical order. However, we want to sort by the count of the word. The `-n` option is used to sort the lines numerically. The `-r` option is used to sort in reverse order.

**Exercise 5.4.1** Try to run the same command without the `-n` option. Observe how the sorting makes sense alphabetically, but not numerically.

```

1 $ tr 'A-Z' 'a-z' < alice_in_wonderland.txt | 
  tr -cd 'a-z ' | tr ' ' '\n' | grep . | 
  sort | uniq -c | sort -nr | head -20
2 1442 the
3 713 and
4 647 to
5 558 a
6 472 she
7 463 it
8 455 of
9 435 said
10 357 i
11 348 alice
12 332 you
13 332 in

```

```

14      313 was
15      241 that
16      237 as
17      202 her
18      190 at
19      169 on
20      161 all
21      158 with

```

Finally, we have the top 20 words in the file `alice_in_wonderland.txt` along with the count of each word.

Although the above command is a single line<sup>10</sup>, it is doing a lot of processing. Still, it is very readable. This is the power of pipes.

10: Such commands are called one-liners.

**Remark 5.4.3** Usually, when we come across such one-liners, it may initially seem too complicated to understand. The key to understanding such one-liners is to break them down into smaller parts and understand them one component at a time from left to right. Feel free to execute each command separately and observe the output like we did above.

### 5.4.3 Piping Standard Error

Pipes are used to connect the standard output of one process to the standard input of another process. However, the standard error is not connected and remains mapped to the terminal. This is because the standard error is a separate stream from the standard output.

However, we can connect the standard error to the standard input of another process using the `2>&1` operator. This is useful when we want to process the error messages of a command.

```

1 $ ls -d /home /nonexistant | wc
2 "/nonexistant": No such file or directory (os
      error 2)
3      1      1      6
4 $ ls -d /home /nonexistant 2>&1 | wc
5      2     10     61

```

This is same as redirecting both the streams to a single file as demonstrated earlier.

However, there is a shorter way to do this using the `&` `|` syntactic sugar.

```

1 $ ls -d /home /nonexistant |& wc
2      2     10     61

```

This does the exact same thing as `2>&1` followed by the pipe.

#### 5.4.4 Piping to and From Special Files

As we discussed earlier, there are some special files in the `/dev` directory.

##### `/dev/null`

The `/dev/null` file is a special file that discards all the data that is written to it. It is like a black hole. All the data that is not needed can be written to this file. Usually errors are written to the `/dev/null` file.

```

1 $ ls -d /home 2> /dev/null
2 /home

```

The error is not actually stored in any file, thus the storage space is not wasted.

##### `/dev/zero`

The `/dev/zero` file is a special file that provides an infinite stream of null bytes. This is useful when

you want to provide an infinite stream of data to a process.

```
1 $ head -c1024 /dev/zero > zero.txt
2 $ ls -lh zero.txt
3 -rw-r--r-- 1 sayan sayan 1.0K Jul  4 15:04
   zero.txt
```

Here we are taking the first 1024 bytes from the `/dev/zero` and writing it to the file `zero.txt`. The file `zero.txt` is 1.0K in size. This is because the `/dev/zero` file provides an infinite stream of null bytes, of which 1024 bytes are taken.

**Warning 5.4.1** Make sure to always use `head` or any other limiter when working with `/dev/zero` or `/dev/random` as they are infinite streams. Forgetting this can lead to the disk being filled up. `Head` with the default parameter will also not work, since it depends on presence of new-line characters, which is not there in an infinite stream of zeros. That is why we are using a byte count limiter using the `-c` option. If you forget to add the limiter, you can press `Ctrl+C` as quickly as possible to stop the process and then remove the file using `rm`.

### **/dev/random and /dev/urandom**

The `/dev/random` and `/dev/urandom` files are special files that are infinite suppliers of random bytes. The `/dev/random` file is a blocking random number generator. This means that it will block if there is not enough entropy. The `/dev/urandom` file is a non-blocking random number generator. This means that it will not block even if there is not enough entropy. Both can be used to generate random numbers.

```

1 | $ head -c1024 /dev/random > random.txt
2 | $ ls -lh random.txt
3 | -rw-r--r-- 1 sayan sayan 1.0K Jul  4 15:04
   |           random.txt

```

Observe that here too the file is of size 1.0K. This is because we are still taking only the first 1024 bytes from the infinite stream of random bytes. However, if we gzip the data, we can see that the zeros file is much smaller than the random file.

```

1 | $ gzip random.txt zero.txt
2 | $ ls -lh random.txt.gz zero.txt.gz
3 | -rw-r--r-- 1 sayan sayan 1.1K Jul  4 15:11
   |           random.txt.gz
4 | -rw-r--r-- 1 sayan sayan   38 Jul  4 15:10
   |           zero.txt.gz

```

The random file is 1.1K in size, while the zero file is only 38 bytes in size. This is because the random file has more entropy and thus cannot be compressed as much as the zero file.

### 5.4.5 Named Pipes

A **named pipe** is a special file that provides a way to connect the standard output of one process to the standard input of another process. This is done by creating a special file in the filesystem. We have already covered named pipes in the Chapter 1 chapter.

Although a pipe is faster than a named pipe, a named pipe can be used to connect processes that are not started at the same time or from the same shell. This is because a named pipe is a file in the filesystem and can be accessed by any process that has the permission to access the file.

Try out the following example. First create a named pipe using the `mkfifo` command.

```
1| $ mkfifo pipe1
```

Then run two processes, one that writes to the named pipe, another that reads from the named pipe. The order of running the processes is not important.

**Terminal 1:**

```
1| $ cat /etc/profile > pipe1
```

**Terminal 2:**

```
1| $ wc pipe1  
2|      47      146      993 pipe1
```

**Exercise 5.4.2** After you have created the named pipe, try changing the order of running the other two processes. Observe that whatever is run first will wait for the other process to start. This is because a named pipe is not storing the data piped to it in the filesystem. It is simply a buffer in the memory.

A named pipe is more useful over regular files when two processes want to communicate with each other. This is because a named pipe is

1. Faster than a regular file as it does not store the data in the file system.
2. Independent of the order of launching the processes. The reader can be launched first and it will still wait for the writer to send the data.
3. Works concurrently. The writer does not need to be done writing the entire data before the reader can start reading. The reader can start

as soon as the writer starts writing. The faster process will simply block until the slower process catches up.

To demonstrate the last point, try running the following commands.

Ensure that a named pipe is created.

#### Terminal 1:

```
1 | $ grep linux pipe1
```

This process is looking for lines containing the word `linux` in the file `pipe1`. Initially it will simply block. Grep is a command that does not wait for the entire file to be read. It starts printing output as soon as a line containing the pattern is read.

#### Terminal 2:

```
1 | $ tree / > pipe1
```

This command is attempting to list out each and every file on the filesystem. This takes a lot of time. However, since we are using a named pipe, the `grep` command will start running as soon as the first line is written to the pipe.

You can now observe the first terminal will start printing some lines containing the word `linux` as soon as the second terminal starts writing to the pipe.

Now try the same with a regular file.

```
1 | $ touch file1 # ensure its a normal file
2 | $ tree / > file1
3 | $ grep linux file1
```

Since this is a regular file, we cannot start reading from the file before the entire file is written. If we do that, the `grep` command will quit as soon as it

catches up with the writer, it will not block and wait.

Observe that to start getting output on the screen takes a lot longer in this method. Not to mention the disk space wasted due to this.

**Remark 5.4.4** Remember that when we use redirection (>) to write to a file, the shell truncates the file. But when we use a named pipe, the shell knows that the file is a named pipe and does not truncate the file.

#### 5.4.6 Tee Command

The tee command is a command that reads from the standard standard input and writes to the standard output and to a file. It is very useful when you want to save the output of a command but also want to see the output on the terminal.

```
1 $ ls -d /home /nonexistent | tee output.txt
2 ls: cannot access '/nonexistent': No such file
   or directory
3 /home
4 $ cat output.txt
5 /home
```

Observe that only the standard output is written to the file, and not the standard error. This is because pipes only connect the standard output to the standard input of the next command, and the standard error remains mapped to the terminal.

Thus in the above output, although it may look like that both the standard output and the standard error are written to the terminal by the same command, it is not so.

The standard error of the ls command remains mapped to the terminal, and thus gets printed directly, whereas the standard output is redirected to the standard input of the tee command, which then prints that to the standard output (and also writes it to the file).

You can also mention multiple files to write to.

```

1 $ ls -d /home | tee output1.txt output2.txt
2 $ cat output1.txt
3 /home
4 $ cat output2.txt
5 /home
6 $ diff output1.txt output2.txt
7 $
```

**Remark 5.4.5** The diff command is used to compare two files. If the files are the same, then the diff command will not print anything. If the files are different, then the diff command will print the lines that are different.

We can also append to the file using the -a option.

```

1 $ ls -d /home | tee output.txt
2 $ ls -d /etc | tee -a output.txt
3 $ cat output.txt
4 /home
5 /etc
```

## 5.5 Command Substitution

We have already seen that we can run multiple commands in a subshell in bash by enclosing them in parentheses.

```

1 $ (whoami; date)
```

```
2 | sayan
3 | Thu Jul 4 09:05:30 AM IST 2024
```

This is useful when you simply want to print the standard output of the commands to the terminal. However, what if you want to store the output of the commands in a variable? Or what if you want to pass the standard output of the commands as an argument to another command?

To do this, we use command substitution. Command substitution is a way to execute one or more processes in a subshell and then use the output of the subshell in the current shell.

There are two ways to do command substitution.

1. Using backticks ‘command’ - this is the legacy way of doing command substitution. It is not recommended to use this as it is difficult to read and can be confused with single quotes. It is also harder to nest.
2. Using the `$(command)` syntax - this is the recommended way of doing command substitution. It is easier to read and nest.

```
1 | $ echo "Today is $(date)"
2 | Today is Thu Jul 4 09:05:30 AM IST 2024
```

Throughout this book, we will use the `$(command)` syntax and not the backticks.

Here we are using the `$(date)` command substitution to get the current date and time and then using it as an argument to the `echo` command.

```
1 | $ myname="$(whoami)"
2 | $ mypc="$(hostname)"
3 | $ echo "Hello, $myname from $mypc"
4 | Hello, sayan from rex
```

We can store the output of the command in a variable and then use it later. This is useful when you want to use the output of a command multiple times.

**Remark 5.5.1** Although you do not need to use the quotes with the command substitution in this case, it is always recommended to use quotes around the variable assignment, since if the output is multiword or multiline, it will throw an error.

## 5.6 Arithmetic Expansion

Arithmetic expansion allows the evaluation of an arithmetic expression and the substitution of the result. The format for arithmetic expansion is:

```
1 | $(( expression ))
```

This is the reason we cannot directly nest subshells without a command between the first and the second subshell.

```
1 | $ cat /dev/random | head -c$((1024*1024)) >
    random.txt
2 | $ ls -lh random.txt
3 | -rw-r--r-- 1 sayan sayan 1.0M Jul  4 15:04
    random.txt
```

11: 1MiB = 1024KiB =  $1024 \times 1024$  bytes - this is called a mebibyte  
 1MB = 1000KB = 1000 × 1000 bytes - this is called a megabyte

This is a very common confusion amongst common people. Kilo, Mega, Giga are SI prefixes, while Kibi, Mebi, Gibi are IEC prefixes.

Here we are using **arithmetic expansion** to calculate the number of bytes in 1MiB<sup>11</sup> and then using it as an argument to the head command. This results in creation of a file random.txt of size 1MiB.

### 5.6.1 Using variables in arithmetic expansion

We can also use variables in arithmetic expansion. We don't have to use the \$ operator to access the

value of the variable inside the arithmetic expansion.

```
1 $ a=10
2 $ b=20
3 $ echo $((a+b))
4 30
```

There are other ways to do arithmetic in bash, such as using the `expr` command, or using the `let` command. However, the `$()` syntax is the most recommended way to do simple arithmetic with variables in bash.

## 5.7 Process Substitution

Process substitution is a way to provide the output of a process as a file. This is done by using the `<(command)` syntax.

Some commands do not accept standard input. They only accept a filename as an argument. This is the exact opposite of the issue we had with the `read` command, which accepted only standard input and not a filename. There we used the `<` operator to redirect the standard input from a file.

The `diff` command is a command that compares two files and prints out differences. It does not accept standard input. If we want to compare differences between the output of two processes, we can use process substitution.

Imagine you have two directories and you want to compare the files in the two directories. You can use the `diff` command to compare the two directories.

```

1 $ ls
2 $ dir1 dir2
3 $ ls dir1
4 file1 file2 file3
5 $ ls dir2
6 file2 file3 file4

```

We can see that the two directories have some common files (`file2` and `file3`) and some different files (`file1` in `dir1` and `file4` in `dir2`).

However, if we have a lot of files, it is difficult to see manually which files are different.

Let us first try to save the output of the two `ls` commands to files and then compare the files using `diff`.

```

1 $ ls
2 dir1 dir2
3 $ ls dir1 > dir1.txt
4 $ ls dir2 > dir2.txt
5 $ diff dir1.txt dir2.txt
6 1d0
7 < file1
8 3a3
9 > file4

```

Great! We can see that the file `file1` is present only in `dir1` and the file `file4` is present only in `dir2`. All other files are common.

However observe that we had to create two files `dir1.txt` and `dir2.txt` to store the output of the `ls` commands. This is not efficient. If the directories contained a million files, then we would have to store tens or hundreds of megabytes of data in the files.

It sounds like a job for the named pipes we learnt earlier. Lets see how easier or harder that is.

```
1 $ ls
2 dir1 dir1.txt dir2 dir2.txt
3 $ rm dir1.txt dir2.txt
4 $ mkfifo dir1 fifo dir2 fifo
5 $ ls dir1 > dir1 fifo &
6 $ ls dir2 > dir2 fifo &
7 $ diff dir1 fifo dir2 fifo
8 1d0
9 < file1
10 3a3
11 > file4
```

Et voila! We have the same output as before, but without the data actually being stored in the filesystem. The data is simply stored in the memory till the `diff` command reads it. However observe that we had to create two named pipes, and also run the `ls` processes in the background as otherwise they would block. Also, we have to remember to delete the named pipes after using them. This is still too much hassle.

Let us remove the named pipes and the files.

```
1 $ rm *fifo
```

Now let us see how easy it is using process substitution.

```
1 $ diff <(ls dir1) <(ls dir2)
2 1d0
3 < file1
4 3a3
5 > file4
```

Amazing! We have the same output as before, but without having to initialize anything. The process substitution does all the magic of creating temporary named pipes and running the processes with the correct redirections concurrently. It then substitutes the filenames of the named pipes in the place of the process substitution.

12: Try to find if this is used in the evaluation scripts of the VM Tasks!

Process substitution is also extremely useful when comparing between expected output and actual output in some evaluation of a student's scripts.

12

We can also use process substitution to provide input to a process running in the subshell.

```
1 | tar cf >(bzip2 -c > file.tar.bz2) folder1
```

This calls `tar cf /dev/fd/?? folder1`, and `bzip2 -c > file.tar.bz2`.

This example is lifted from <https://tldp.org/LDP/abs/html/process-sub.html>.

If you are interested in more examples of process substitution, refer the same.

Because of the `/dev/fd/<n>` system feature, the pipe between both commands does not need to be named. This can be emulated as

```
1 | mkfifo pipe
2 | bzip2 -c < pipe > file.tar.bz2 &
3 | tar cf pipe folder1
4 | rm pipe
```

**Remark 5.7.1** `tar` is a command that is used to create archives. It simply puts all the files and directories in a single file. It does not perform any compression. The `c` option is used to create an archive. The `f` option is used to mention the name of the archive. The `bzip2` command is used to compress files. The `-c` option is used to write the compressed data to the standard output. The `>` operator is used to redirect the standard output to a file. We will cover `tar` and `zips` in more detail later.

That is pretty much all you need to know about pipes and redirections. To really understand and appreciate the power of pipes and redirections, you have to stop thinking imperically (like C or Python) and start thinking in streams, like a functional programming language. Once this paradigm shift

happens, you will start to see the power of pipes and redirections and will be able to tackle any kind of task in the command line.

## 5.8 Summary

Let us quickly summarize the important syntax and commands we learnt in this chapter.

**Table 5.1:** Pipes, Streams, and Redirection syntax

Syntax	Command	Description
;	Command Separator	Run multiple commands in a single line
&&	Logical AND	Run the second command only if the first command succeeds
	Logical OR	Run the second command only if the first command fails
>	Output Redirection	Redirect the stdout of the process to a file
>>	Output Redirection	Append the stdout of the process to a file
<	Input Redirection	Redirect the stdin of the process from a file
2>	Error Redirection	Redirect the stderr of the process to a file
2>\&1	Error Redirection	Redirect the stderr of the process to the stdout
<<EOF	Here Document	Redirect the stdin of the process from a block of text
<<	Here String	Redirect the stdin of the process from a string
	Pipe	Connect the stdout of one process to the stdin of another process
&	Pipe Stderr	Connect the stderr of one process to the stdin of another process
\$(command)	Command Substitution	Run a command and use the output in the current shell
\$((expression))	Arithmetic Expansion	Evaluate an arithmetic expression
<(command)	Process Substitution	Provide the output of a process as a file
>(command)	Process Substitution	Provide the input to a process from a file



# 6

# Pattern Matching

## 6.1 Introduction

We have been creating files and directories for a while now, and we have often required to search for files or directories in a directory. Till now we used to use the `ls` command to list out all the files in a directory and then check if the file we are looking for is present in the list or not. This works fine when the number of files is less, but when the number of files is large, this method becomes cumbersome. This is where we can use pattern matching to search for files or directories or even text in a file.

You would have also used the popular `Ctrl+F` shortcut on most text editors or browsers to search for text in a file or on a webpage. This is also an example of pattern matching.

## 6.2 Globs and Wildcards

The simplest form of pattern matching is using glob for filename expansion. Globs are used to match filenames in the shell.

**Definition 6.2.1 (Glob)** A glob is a pattern-matching mechanism used for filename expansion in the shell. The term "glob" represents the concept of matching patterns globally or expansively across multiple filenames or paths.

In bash, we can use the following wildcards to match filenames:

- ▶ \* - Matches zero or more characters.
- ▶ ? - Matches exactly one character.
- ▶ [abc] - Matches any one of the characters within the square brackets.
- ▶ [a-z] - Matches any one of the characters in the range.
- ▶ [!abc] - Matches any character except the ones within the square brackets.

Let us explore these in detail.

Try to guess the output of each of the command before seeing the output. If you get an output different from what you expected, try to understand why.

```

1 $ touch abc bbc zbc aac ab
2 $ ls -1
3 aac
4 ab
5 abc
6 bbc
7 zbc
8 $ echo a*
9 aac ab abc
10 $ echo a?
11 ab
12 $ echo ?bc
13 abc bbc zbc
14 $ echo [ab]bc
15 abc bbc
16 $ echo [az]bc
17 abc zbc
18 $ echo [a-z]bc
19 abc bbc zbc
20 $ echo [!ab]bc
21 zbc
22 $ echo [!z]bc
23 abc bbc
24 $ echo [!x-z]?c
25 aac abc bbc

```

Shell globs only work with files and directories in the current directory. The glob expansion to sorted list of valid files in the current directory is done by the shell, and not by the command itself. It is done

before the command is executed. The command thus does not even know that a glob was used to expand the filenames. To the command, it looks like the user directly typed the filenames.

A glob always expands to a space separated list of filenames. However, how the command interprets this list of filenames is up to the command. Some commands such as `ls -1` will print each filename on a new line, whereas some commands such as `echo` will print all filenames on the same line separated by a space. `echo` does not care if the arguments passed to it are filenames or not. It just prints them as is.

```
1 $ echo a*
2 aac ab abc
3 $ ls -1 a*
4 aac
5 ab
6 abc
7 $ ls a*
8 aac ab abc
9 $ wc a*
10 0 0 0 aac
11 0 0 0 ab
12 0 0 0 abc
13 0 0 0 total
14 $ stat a*
15   File: aac
16   Size: 0           Blocks: 0          IO
        Block: 4096  regular empty file
17 Device: 8,2      Inode: 4389746    Links: 1
18 Access: (0644/-rw-r--r--) Uid: ( 1000/
        sayan) Gid: ( 1001/  sayan)
19 Access: 2024-07-12 19:10:27.542322238 +0530
20 Modify: 2024-07-12 19:10:27.542322238 +0530
21 Change: 2024-07-12 19:10:27.542322238 +0530
22 Birth: 2024-07-12 19:10:27.542322238 +0530
23   File: ab
24   Size: 0           Blocks: 0          IO
```

```

      Block: 4096    regular empty file
25 Device: 8,2      Inode: 4389748      Links: 1
26 Access: (0644/-rw-r--r--) Uid: ( 1000/
               sayan)  Gid: ( 1001/  sayan)
27 Access: 2024-07-12 19:16:10.707221331 +0530
28 Modify: 2024-07-12 19:16:10.707221331 +0530
29 Change: 2024-07-12 19:16:10.707221331 +0530
30 Birth: 2024-07-12 19:16:10.707221331 +0530
31 File: abc
32 Size: 0          Blocks: 0           IO
      Block: 4096    regular empty file
33 Device: 8,2      Inode: 4389684      Links: 1
34 Access: (0644/-rw-r--r--) Uid: ( 1000/
               sayan)  Gid: ( 1001/  sayan)
35 Access: 2024-07-12 19:10:22.055523865 +0530
36 Modify: 2024-07-12 19:10:22.055523865 +0530
37 Change: 2024-07-12 19:10:22.055523865 +0530
38 Birth: 2024-07-12 19:10:22.055523865 +0530

```

As seen above, the globs simply expand to the filenames in the current path and pass it as arguments to the command. The output of the command depends on what command it is. The `wc` command counts the number of lines, words, and characters in a file. The `stat` command prints out the metadata of the files. Similarly, we can use the `file` command to print the type of the files. Try it out.

**Exercise 6.2.1** Go to an `empty` directory, and run the following command.

```

1 |   $ expr 5 * 5
2 |

```

Now run the following command.

```

1 |   $ touch +
2 |   $ expr 5 * 5
3 |

```

Observe the output of the commands. Can you explain why the output is different?

When using globs in the shell, if a glob does not match any files, it is passed as is to the command. The command then interprets the glob as a normal string.

```
1 $ touch abc bbc
2 $ ls
3 abc  bbc
4 $ echo ?bc
5 abc  bbc
6 $ echo ?bd
7 ?bd
```

As echo does not care if the arguments passed to it are filenames or not, it simply prints the arguments as is. The ls command, however, will not print the filenames if they do not exist and will instead print to the standard error that the file does not exist. Use this knowledge to decipher why the above exercise behaves the way it does.

## 6.3 Regular Expressions

Globs are good enough when we simply want to run some command and pass it a list of files matching some pattern. However, when we want to do more complex pattern matching, we need to use regular expressions.

**Definition 6.3.1** (Regular Expression) A regular expression (shortened as regex or regexp), sometimes referred to as rational expression, is a sequence of characters that specifies a match pattern in text. Usually such patterns are used by string-searching algorithms for "find" or "find and replace" operations on strings, or for input

validation. Regular expression techniques are developed in theoretical computer science and formal language theory.

Due to the powerfullness of regular expressions, almost all programming languages and text processing tools support regular expressions directly. This makes text processing very easy and powerful, as well as cross-platform.

However, there are multiple flavors of regular expressions, and the syntax of each flavor may differ slightly. The most common flavors are:

- ▶ Basic Regular Expressions (BRE)
- ▶ Extended Regular Expressions (ERE)
- ▶ Perl-Compatible Regular Expressions (PCRE)

These are the regular expressions that are supported by most text processing utilities in Unix-like systems. These follow the POSIX standard for regular expressions. There are also Perl syntax regular expressions, which are more powerful and flexible, and are supported by the Perl programming language.<sup>1</sup>

We will focus on BRE and ERE in this chapter, as these are the most commonly used flavors in Unix-like systems.

1: PCRE and Perl Regex are not the same. PCRE is a library that implements Perl like regular expressions, but in C. Perl Regex is the regular expression that is used in the Perl programming language. More details can be found online.

2: awk uses ERE by default, not BRE.

### 6.3.1 Basic Regular Expressions

Basic Regular Expressions (BRE) are the simplest form of regular expressions. They are supported by most Unix-like systems and are the default regular expressions used by most text processing utilities such as grep, and sed.<sup>2</sup>

BRE syntax is similar to the glob syntax, but with more power and flexibility. There are some subtle differences between the two.

The following are the basic regular expressions that can be used in BRE:

- ▶ a - Matches the character a.
- ▶ . - Matches any single character exactly once.
- ▶ \* - Matches zero or more occurrences of the previous character.
- ▶ ^ - Matches the null string at the start of a line.
- ▶ \$ - Matches the null string at the end of a line.
- ▶ [abc] - Matches any one of the characters within the square brackets.
- ▶ [a-z] - Matches any one of the characters in the range, both ends inclusive.
- ▶ [^abc] - Matches any character except the ones within the square brackets; the caret symbol has a different meaning when inside the brackets.
- ▶ \+ - Matches one or more occurrences of the previous character.
- ▶ \? - Matches zero or one occurrence of the previous character.
- ▶ {n} - Matches exactly n occurrences of the previous character.
- ▶ {n,} - Matches n or more occurrences of the previous character.
- ▶ {n,m} - Matches n to m occurrences of the previous character.
- ▶ \ - Escapes a special character such as \*, ., [, \, \$, or ^.
- ▶ regex1|regex2 - Matches either regex1 or regex2.
- ▶ (regex) - Groups the regex.
- ▶ \2 - Matches the 2-nd (...) parenthesized subexpression in the regular expression. This

is called a back reference. Subexpressions are implicitly numbered by counting occurrences of `(` left-to-right.

- `\n` - Matches a newline character.

### 6.3.2 Character Classes

A bracket expression is a list of characters enclosed by `[` and `]`. It matches any single character in that list; if the first character of the list is the caret `^`, then it matches any character not in the list. For example, the following regex matches the words `'gray'` or `'grey'`.

```
gr[ae]y
```

Let's create a small script to test this regex.

```

1 $ cat regex1.sh
2 #!/bin/bash
3 read -r -p "Enter color: " color
4 if [[ $color =~ gr[ae]y ]]; then
5     echo "The color is gray or grey."
6 else
7     echo "The color is not gray or grey."
8 fi
9 $ ./regex1.sh
10 Enter color: gray
11 The color is gray or grey.
12 $ ./regex1.sh
13 Enter color: grey
14 The color is gray or grey.
15 $ ./regex1.sh
16 Enter color: green
17 The color is not gray or grey.

```

## Ranges

Within a bracket expression, a range expression consists of two characters separated by a hyphen. It matches any single character that sorts between the two characters, inclusive. In the default C locale, the sorting sequence is the native character order; for example, '[a-d]' is equivalent to '[abcd]'.<sup>3</sup>

For example, the following regex matches any lowercase letter.

```
[a-z]
```

Let's create a small script to test this regex.

```

1 $ cat regex2.sh
2 #!/bin/bash
3 read -r -p "Enter a letter: " letter
4 if [[ $letter =~ [a-z] ]]; then
5     echo "The letter is a lowercase letter."
6 else
7     echo "The letter is not a lowercase letter
8 ."
9 fi
10 $ ./regex2.sh
11 Enter a letter: a
12 The letter is a lowercase letter.
13 $ ./regex2.sh
14 Enter a letter: A
15 The letter is not a lowercase letter.
```

3: There are locales other than the default C locale, such as the en\_US.UTF-8 locale, which sorts and collates characters differently. In the en\_US.UTF-8 locale, the sorting sequence is based on the Unicode code points of the characters and collates characters with accents along with the characters.

## Named Character Classes

There are some predefined character classes which are used often, that can be used in regular expressions. These classes contain a pair of brackets, and should be present inside a bracket expression. Some of the common character classes are:

- ▶ `[:alnum:]` - Alphanumeric characters: `[:alpha:]` and `[:digit:]`; in the ‘C’ locale and ASCII character encoding, this is the same as `[0-9A-Za-z]`.
- ▶ `[:alpha:]` - Alphabetic characters: `[:lower:]` and `[:upper:]`; in the ‘C’ locale and ASCII character encoding, this is the same as `[A-Za-z]`.
- ▶ `[:blank:]` - Blank characters: space and tab.
- ▶ `[:cntrl:]` - Control characters. In ASCII, these characters have octal codes 000 through 037, and 177 (DEL). In other character sets, these are the equivalent characters, if any.
- ▶ `[:digit:]` - Digits: 0 1 2 3 4 5 6 7 8 9.
- ▶ `[:graph:]` - Graphical characters: `[:alnum:]` and `[:punct:]`.
- ▶ `[:lower:]` - Lower-case letters; in the ‘C’ locale and ASCII character encoding, this is a b c d e f g h i j k l m n o p q r s t u v w x y z.
- ▶ `[:print:]` - Printable characters: `[:alnum:]`, `[:punct:]`, and space.
- ▶ `[:punct:]` - Punctuation characters.
- ▶ `[:space:]` - Space characters: in the ‘C’ locale, this is tab, newline, vertical tab, form feed, carriage return, and space.
- ▶ `[:upper:]` - Upper-case letters: in the ‘C’ locale and ASCII character encoding, this is A B C D E F G H I J K L M N O P Q R S T U V W X Y Z.
- ▶ `[:xdigit:]` - Hexadecimal digits: 0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f.

These named character classes’s expansion depends on the locale. For example, in the en\_US.UTF-8 locale, the `[:lower:]` class will match all lowercase letters in the Unicode character set, not just the ASCII character set.

It is important to note that these named character classes should be present inside two square brackets, and not just one. If we use only one square bracket, it will interpret each character inside the square brackets as a separate character in the list.

`[:digit:]` will match any of the characters `d g i t ..`. If a character is repeated in the list, it has no additional effect.

Some characters have different meaning inside the list depending on the position they are in. For example, the caret symbol `^` negates the entire list if it is the first character in the list, but is matched literally if it is not the first character in the list.

- ▶ `]` is used to end the list of characters, unless it is the first character in the list, then it is matched literally.
- ▶ `-` is used to specify a range of characters, unless it is the first or last character in the list, then it is matched literally.
- ▶ `^` is used to negate the list of characters if it is the first character in the list, else it is matched literally.

Example for `]:`

```
1 $ echo "match square brackets [ and ]" | grep
   -o '[mb]'
2 m
3 b
4 $ echo "match square brackets [ and ]" | grep
   -o '[]mb'
5 m
6 b
7 ]
```

Example for `-:`

```
1 $ echo "ranges are separated by hyphens like -"
   " | grep -o '[a-c]'
```

From this example, we have started using the `grep` command to match regex quickly instead of creating a script. We will discuss the `grep` command in more detail later in this chapter. The `-o` flag is used to print only the matching part of the line, instead of the entire line. If you omit it, and run the command, you will see the entire line that has the matching part being printed, however the matching part may still be highlighted using color. As it is not possible to highlight the matching part in this book, we are using the `-o` flag to print only the matching part.

Observe how putting the hyphen at the start of the list makes it match the hyphen literally, whereas putting it in the middle makes it match a range of characters.

```

2 a
3 a
4 a
5 a
6 b
7 $ echo "ranges are separated by hyphens like -
           " | grep -o '[-a-c]'
8 a
9 a
10 a
11 a
12 b
13 -

```

The caret symbol `^` is used to negate the list of characters if it is the first character in the list, else it is matched literally. First case will match any character except `l`, `i`, `n`, `e`, and the second case will match only the characters `l`, `i`, `n`, `e` and the caret symbol `^`.

Example for `^`:

```

1 $ echo "this is a ^line^" | grep -o '[^line]'
2 t
3 h
4 s
5 s
6 s
7
8 a
9
10 ^
11 ^
12 $ echo "this is a ^line^" | grep -o '[line^]'
13 i
14 i
15 ^
16 l
17 i
18 n
19 e
20 ^

```

## Collating Symbols

A collating symbol is a single-character collating element enclosed in '[.' and '.]'. It stands for a collating element that collates with a single character, as if the character were a separate character in the POSIX locale's collation order. Collating symbols are typically used when a digraph is treated like a single character in a language. They are an element of the POSIX regular expression specification, and are not widely supported.

For example, the Welsh alphabet<sup>4</sup> has a number of digraphs that are treated as a single letter (marked with a \* below)

1	a b c ch d dd e f ff g ng h i j l ll m n o p
	ph r rh s t th u w y
2	* * * *
	* * *

Assuming the locale file defines it<sup>5</sup>, the collating symbol [[.ng.]] is treated like a single character. Likewise, a single character expression like . or [^a] will also match "ff" or "th." This also affects sorting, so that [p-t] will include the digraphs "ph" and "rh" in addition to the expected single letters.

A collating symbol represents a set of characters which are considered as a single unit for collating (sorting) purposes; for example, "ch"/"Ch" or "ss" (these are only valid in locales which define them);

4: Read more about the Welsh alphabet [here](#).

5: a collating symbol will only work if it is defined in the current locale

## Equivalence Classes

An equivalence class groups characters which are equivalent for collating purposes; for example, "a" and "à" (and other accented variants).

`[[:=a=]]` is an equivalence class that matches the character "a" and all its accented variants, such as `a³àâãäå`, etc.

Collating symbols and equivalence classes are used in locale definitions to encode complex ordering information and are not implemented in some regular expression engines. We will not discuss these in depth.

## Escape Sequences

Along with the named character classes, there are some escape sequences that can be used in regular expressions. These are:

- ▶ `\b` - Matches a word boundary; that is it matches if the character to the left is a “word” character and the character to the right is a “non-word” character, or vice-versa. It does not match any character, but matches the empty string that marks the word delimitation.
- ▶ `\B` - Matches a non-word boundary; that is it matches if the characters on both sides are either “word” characters or “non-word” characters.
- ▶ `\<` - Matches the start of a word only.
- ▶ `\>` - Matches the end of a word only.
- ▶ `\d` - Matches a digit character. Equivalent to `[0-9]`.
- ▶ `\D` - Matches a non-digit character. Equivalent to `[^0-9]`.
- ▶ `\s` - Matches a whitespace character. Equivalent to `[:space:]`.
- ▶ `\S` - Matches a non-whitespace character. Equivalent to `[^[:space:]]`.
- ▶ `\w` - Matches a word character. Equivalent to `[:alnum:]_`.

- ▶ \W - Matches a non-word character. Equivalent to `[^[:alnum:]_]`.
- ▶ \' - Matches the start of pattern space if multiline mode is enabled.
- ▶ \' - Matches the end of pattern space if multiline mode is enabled.

Other than these, other escape characters are present to match special non-graphical characters such as newline, tab, etc. These are GNU extensions and are not defined in the original POSIX standard.

- ▶ \a - Matches the alert character (ASCII 7).
- ▶ \f - Matches the form feed character (ASCII 12).
- ▶ \n - Matches the newline character (ASCII 10).
- ▶ \r - Matches the carriage return character (ASCII 13).
- ▶ \t - Matches the tab character (ASCII 9).
- ▶ \v - Matches the vertical tab character (ASCII 11).
- ▶ \0 - Matches the null character (ASCII 0).
- ▶ \cx - Matches the control character x. For example, \cM matches the carriage return character. This converts lowercases to uppercase then flips the bit-6 of the character.
- ▶ \xxx - Matches the character with the hex value xx.
- ▶ \oxxx - Matches the character with the octal value xxx.
- ▶ \dxx - Matches the character with the decimal value xxx.

You can also read more about the locale issues in regex [here](#).

### 6.3.3 Anchors

Anchors are used to match a position in the text, rather than a character. The following are the anchors that can be used in regular expressions:

- ▶ ^ - Matches the start of a line.
- ▶ \$ - Matches the end of a line.
- ▶ \b - Matches a word boundary.
- ▶ \B - Matches a non-word boundary.
- ▶ \< - Matches the start of a word.
- ▶ \> - Matches the end of a word.
- ▶ \` - Matches the start of the pattern space if multiline mode is enabled.
- ▶ \' - Matches the end of the pattern space if multiline mode is enabled.

These anchors are used to match the position in the text, rather than the character. For example, the regex `^a` will match the character `a` only if it is at the start of the line. It does not match any other character other than the `a`. However, if there are not a present at the start of the line, then nothing is matched at all.

`xxd` is a command that is used to convert a file to a hex dump. It is used here to show the output in a more readable format. `61` is the hex value of the character `a`. `0a` is the hex value of the newline character. The lack of output in the second case means that nothing is matched, and no bytes are output.

```

1 $ echo "apple" | grep -o '^a'
2 a
3 $ echo "apple" | grep -o '^a' | xxd
4 00000000: 610a
5           a.
6 $ echo "banana" | grep -o '^a'
6 $ echo "banana" | grep -o '^a' | xxd

```

The anchors `^`, `$`, and `\b` are very useful in most of the text processing tasks. The `^` and `$` when used together can match the entire line, meaning that the pattern between them is not matched if it is a substring of the line; it only matches if the pattern is the entire line. The `\b` is used to surround the

pattern if we want to match the pattern as a word, and not as substring of a word.

```

1 $ echo "apple" | grep -o '^apple$'
2 apple
3 $ echo "apple is great" | grep -o '^apple$'
4 $ echo "apple is great" | grep -o '\bapple\b'
5 apple
6 $ echo "i like pineapple" | grep -o '\bapple\b'
,
```

Observe that even though we are using the `-o` flag, the entire word is printed in a single line. This is because the `-o` flag prints only the matches and prints them on separate lines. However, unlike the previous cases where we were using character lists, here the entire word is a single match, and thus is printed on a single line.

### 6.3.4 Quantifiers

Quantifiers are used to match a character or a group of characters multiple times. This is useful if we do not know the exact number of times a character or group of characters will be repeated or its length. Paired with a character list, it makes regex very powerful and able to match any arbitrary pattern.

The following are the quantifiers that can be used in regular expressions:

- ▶ `*` - Matches zero or more occurrences of the previous character.
- ▶ `\+` - Matches one or more occurrences of the previous character.
- ▶ `\?` - Matches zero or one occurrence of the previous character.
- ▶ `{n}` - Matches exactly n occurrences of the previous character.

- ▶  $\{n\}$  - Matches n or more occurrences of the previous character.
- ▶  $\{,n\}$  - Matches n or less occurrences of the previous character.
- ▶  $\{n,m\}$  - Matches n to m occurrences of the previous character, both ends inclusive.

Note that + and ? are not part of the BRE standard, but are part of the ERE standard. However, most text processing utilities support them in BRE mode as well if escaped.

```

1 $ echo -n "aaaaaaaaaaaaaa" | wc -c # there are
   14 a's
2 14
3 $ echo "aaaaaaaaaaaaaa" | grep -E "a{4}" -o #
   the first 12 a's are matched in groups of
   four
4 aaaa
5 aaaa
6 aaaa
7 $ echo "aaaaaaaaaaaaaa" | grep -E "a{4,}" -o #
   entire string is matched
8 aaaaaaaaaaaaaaa
9 $ echo "aaaaaaaaaaaaaa" | grep -E "a{4,5}" -o
   # maximal matching, first 10 a's are
   matched as groups of 5, then the last 4 a'
   s are matched as group of four.
10 aaaaa
11 aaaaa
12 aaaa
13 $ echo "aaaaaaaaaaaaaa" | grep -E "a{,5}" -o
14 aaaaa
15 aaaaa
16 aaaa

```

Here we are using the -E flag to enable ERE mode in grep.

If we are using the default BRE mode, we need to escape the +, ?, {, and }, (, ), | characters to use them.

Let us also see how the \*, + and ? quantifiers work.

```

1 $ echo "There are there main quantifiers,
   which are asterisk (*), plus (+), and
   eroteme (?)."
2 T | grep "[^aeiou][aeiou]*" -o

```

```
3 he
4 re
5 a
6 re
7
8 t
9 he
10 re
11
12 mai
13 n
14
15 qua
16 n
17 ti
18 fie
19 r
20 s
21 ,
22
23 w
24 hi
25 c
26 h
27 a
28 re
29 a
30 s
31 te
32 ri
33 s
34 k
35
36 (
37 *
38 )
39 ,
40
41 p
42 lu
43 s
```

```

44 |
45 | (
46 | +
47 | )
48 | ,
49 | a
50 | n
51 | d
52 | e
53 | ro
54 | te
55 | me
56 |
57 | (
58 | ?
59 | )
60 | .

```

This shows that the asterisk quantifier matches zero or more occurrences of the previous character, here we are matching for any pattern which does not start with a vowel and has zero or more vowels after it, thus the matching can keep on growing as long as there are consecutive vowels. As soon as a non-vowel is present, the previous match ends and a new match starts.

Now compare and contrast the previous output to the next output using the plus quantifier. The lines with only a single character (non-vowel) will no longer be present.

```

1 $ echo "There are three main quantifiers,
      which are asterisk (*), plus (+), and
      eroteme (?)." | grep "[^aeiou][aeiou]\+"
      -
      o
2 he
3 re
4 a
5 re
6 he

```

```
7 re
8 mai
9 qua
10 ti
11 fie
12 hi
13 a
14 re
15 a
16 te
17 ri
18 lu
19 a
20 e
21 ro
22 te
23 me
```

Finally, observe how using the eroteme quantifier will bring back the single character lines, but remove the lines with more than a vowel.

```
1 $ echo "There are there main quantifiers,
      which are asterisk (*), plus (+), and
      eroteme (?)." | grep "[^aeiou][aeiou]\?" -
      o
2 T
3 he
4 re
5 a
6 re
7
8 t
9 he
10 re
11
12 ma
13 n
14
15 qu
16 n
```

```
17 ti
18 fi
19 r
20 s
21 ,
22
23 w
24 hi
25 c
26 h
27 a
28 re
29 a
30 s
31 te
32 ri
33 s
34 k
35
36 (
37 *
38 )
39 ,
40
41 p
42 lu
43 s
44 (
45 +
46 )
47 ,
48 a
49 n
50 d
51 e
52 ro
53 te
54 me
55
56
57 (
```

```

58 ?  

59 )  

60 .

```

When mixed with character lists, quantifiers can be used to match any arbitrary pattern. This makes regular expressions very powerful and flexible.

```

1 $ echo "sometimes (not always) we use  

      parentheses (round brackets) to clarify  

      some part of a sentence (or phrase)." |  

      grep "([^\)]\+)" -o  

2 (not always)  

3 (round brackets)  

4 (or phrase)

```

Observe how `([^\)]\+)` matches any pattern that starts with an opening parenthesis, followed by one or more characters that are not a closing parenthesis, and ends with a closing parenthesis. This lets us match all of the bracketed parts of a sentence, without knowing how many such brackets exist, or what is the length of each expression. This is pretty powerful, and can be used in similar situations, such as extracting text from HTML tags<sup>6</sup> JSON strings, etc.

### 6.3.5 Alternation

Alternation is used to match one of the multiple patterns. It is used to match multiple patterns in a single regex. The syntax for alternation is `regex1 | regex2`. The regex will match if either `regex1` or `regex2` is matched.

Alternation in BRE needs to be escaped, as it is not part of the standard. However, most text processing utilities support it in BRE mode if escaped.

<sup>6</sup>: Regular Expressions can only match regular languages, and not context-free languages, context-sensitive languages, or unrestricted languages. This means that they cannot be used to parse HTML or XML files. However, for simple tasks such as extracting text from tags, regular expressions can be used. To explore community lore on this topic, see [this stackoverflow answer](#). To learn more about the theoretical aspects of regular expressions, see [Chomsky Hierarchy](#) in Theory of Computation.

```

1 $ echo -e "this line starts with t\nand this
      starts with a\nwhereas this line starts
      with w" | grep '^t'
2 this line starts with t
3 $ echo -e "this line starts with t\nand this
      starts with a\nwhereas this line starts
      with w" | grep '^t\|^a'
4 this line starts with t
5 and this starts with a

```

As seen above, the regex `^t\|^a` matches any line that starts with either `t` or `a`. This is very useful when we want to match multiple patterns in a single regex. Note that we have to mention the start of line anchor both times, this is because alternation has the lowest precedence, and thus the start of line anchor is not shared between the two patterns.

Let us now see a more complex example of alternation similar to previous example of brackets.

```

1 $ echo "sometimes (not always) we use
      parentheses (round brackets) or brackets [
      square brackets] to clarify some part of a
      sentence (or phrase)." | grep "([^\)]+)\)
      \|\\([^\)]+\)\" -o
2 (not always)
3 (round brackets)
4 [square brackets]
5 (or phrase)

```

Here we are matching phrases inside round OR square brackets. Observe a few things here:

1. We need to escape the alternation operator `|` as it is not part of the BRE standard.
2. We need to escape the square brackets `[]` when we want it to match literally as they have special meaning in regex.
3. We need to escape the plus quantifier `+` as it is not part of the BRE standard.

### 6.3.6 Grouping

Grouping is used to group multiple characters or patterns together. This is useful when we want to apply a quantifier to multiple characters or patterns. The syntax for grouping is (regex). The regex will match if the pattern inside the parentheses is matched. The parenthesis will not be matched. However, grouping is not present unescaped in BRE, so if we want to match literal parenthesis then we use (regex), and if we want to group the regex without matching the parenthesis, then we use (regex).

Let's revisit one of the earlier examples of alternation, and group the patterns inside the alternation.

```

1 $ echo -e "this line starts with t\nand this
2     starts with a\nwhereas this line starts
3     with w" | grep '^t\|a'
4
5 this line starts with t
6 and this starts with a
7
8 $ echo -e "this line starts with t\nand this
9     starts with a\nwhereas this line starts
10    with w" | grep '^\\(t\\|a\\)'
11
12 this line starts with t
13 and this starts with a

```

As evident from above, both the grouped and ungrouped regexes match the same lines. But in the grouped version, we do not have to repeat the start of line anchor, and the regex is more readable. Also, grouping is useful when we want to apply a quantifier to the entire group.

```

1 $ grep -E "([b-d][f-h][j-n][p-t][v-z])\{2\}"
2     -o <<< "this is a sentence"
3
4 th
5 nt
6 nc

```

Notice the subtly different way of providing the string to the stdin of the `grep` command. We have covered here-strings earlier.

In this example, we are matching any two consecutive characters that are consonants. Here we are not matching **not vowels**, rather we are explicitly matching consonants which are lowercase. Thus this will not match spaces, digits, or punctuations. There is no direct way to match consonants in BRE, so we have to list them explicitly and chain them using alternations. However, if we want to match two consonants consecutively we do not have to list the entire pattern again, we can simply group it and apply the  $\{n\}$  quantifier on it.

The biggest use-case of grouping is to refer to the matched group later in the regex. This is called backreferencing, and is very useful when we want to match a pattern that is repeated later in the text.

```

1 $ grep -E "([b-d]|[f-h]|[j-n]|[p-t]|[v-z])\{2\}"
      -o <<< "this is an attached sentence"
2 th
3 tt
4 ch
5 nt
6 nc

```

Observe in this similar example, where the input string now has the word **attached** in it. One of the matched pattern is **tt**. But if we want to **only** list those consonants groups that use the same consonant, like **tt**? Then we require to use **backreferencing**.

### 6.3.7 Backreferences

Backreferences are used to refer to a previously matched group in the regex. This is useful when we want to match a pattern that is repeated later in the text. The syntax for backreferencing is  $\backslash n$ , where  $n$  is the number of the group that we want to refer

to. The group is implicitly numbered by counting occurrences of (...) left-to-right.

To accomplish the previous example, we use the backreference \1 to match only the same consonant, and not repeat the match using {n}.

```
1 $ grep -E "([b-d]|[f-h]|[j-n]|[p-t]|[v-z])\1"
2      -o <<< "this is an attached sentence"
2 tt
```

Backreferences are also useful in tools such as sed and awk where we can replace the string with another string and can use the matched group and in the replacement string.

For example, if we want to make the first letter of either apple or banana uppercase, we can use the following command.

```
1 $ echo "apple & banana" | sed -E 's/\<([ab])/\
2      U\1/g'
2 Apple & Banana
```

Here we are using the sed command to replace the matched pattern with the uppercase version of the first character. The \1 is used to refer to the first matched group, which is the first character of the word. The syntax of sed is s/pattern/replacement flags, where pattern is the regex to match, replacement is the string to replace the matched pattern with, and flags are the flags to apply to the regex. The replacement string has to be a string, and not a regex, however, we can use backreferences in the replacement string. The \U is used to convert the matched group to uppercase. The g flag is used to replace all occurrences of the pattern in the line. The \< is used to match the start of the word, otherwise the a inside banana will also be capitalized. We will cover sed in details in later chapters.

7: If your distribution does not have the /usr/share

/dict/words file, you can 1 download it from [here](#).

Here we are preprocessing 2 the dictionary to convert all 3 the words to lowercase us- 4 ing the tr command. Then 5 we are passing the output of 6 tr to grep to find the lines 7 that match the pattern ^([ 8 a-z]) [a-z]1\$. This pat- 9 tern matches any three let- 10 ter palindrome. The ^ and \$ 11 are used to match the start 12 and end of the line respec- 13 tively. The ([a-z]) is used 14 to match any lowercase letter 15 and the 1 is used to match 16 the same letter as the first 17 matched group. This is used 18 to match the palindrome. Fi- 19 nally, as the output is too 20 large, we are using the tail 21 command to print only the 22 last 50 lines. We have used 23 process substitution as well 24 as pipes in this, so the flow of 25 data is not strictly left to right. 26 Revise the previous chapters 27 and try to understand how 28 the data is flowing. What dif- 29 ference would it make if we replaced <(tr with < <(tr 30 in the internal workings and 31 the output of the command? 32

Let us use backreferences to find three letter palin-  
dromes in a string. You should have a dictionary of  
words in /usr/share/dict/words.<sup>7</sup>

```
$ grep -E "^(a-z)(a-z)\1$" <(tr 'A-Z' 'a-z'  
    < /usr/share/dict/words ) | tail -n30  
rsr  
rtr  
sas  
sbs  
scs  
sds  
ses  
sis  
sls  
sms  
sos  
sps  
srs  
sss  
sts  
sus  
svs  
sws  
sxs  
tat  
tct  
tet  
tft  
tgt  
tit  
tyt  
tkt  
tnt  
tot  
tpt  
trt  
tst  
tut  
twt  
txt
```

```
37 ulu
38 umu
39 upu
40 uru
41 usu
42 utu
43 vav
44 viv
45 waw
46 wnw
47 wow
48 wsw
49 xix
50 xxx
51 zzz
```

## 6.4 Extended Regular Expressions

Throughout the chapter, we have noticed that some regex syntax are not really supported in BRE standard, and to use them in most text processing applications when using BRE, we have to escape them. Explicitly, the characters that are not supported in BRE but are supported in ERE are as follows.

- ▶ + - In BRE, it would match the literal plus sign if not escaped. Otherwise it is a quantifier of the previous character or group, making it one or more.
- ▶ ? - In BRE, it would match the literal eroteme sign if not escaped. Otherwise it is a quantifier of the previous character or group, making it zero or one, not more.
- ▶ ( and ) - The parenthesis match literal parenthesis in the data in BRE if unescaped, otherwise are used to group regular expressions.

- ▶ { and } - The curly braces match literal curly braces in the data in BRE if unescaped, otherwise are used to specify the number of times the previous character or group is repeated.
- ▶ | - The pipe symbol matches the literal pipe symbol in the data in BRE if unescaped, otherwise is used for alternation.

To use these seven characters with their special meaning directly, without escaping, we can use extended regular expressions.

**Definition 6.4.1 (POSIX-Extended Regular Expressions)** Extended regular expressions (EREs) are a variant of regular expressions that support additional features and syntax. EREs are supported by several command line utilities in Linux, including grep, sed, and awk.

8: as defined by POSIX-ERE standard

In cases where we want to use these symbols for their special meaning<sup>8</sup> instead of as a literal character, we can use the -E flag in grep to enable ERE mode. This will allow us to use these symbols without escaping them. This makes the regular expression easier to read and understand. This is useful since it is less likely that we want to match these symbols literally and more likely that we want to use them for their special meaning.

However, in cases where we want to match these symbols literally, we can escape them using the backslash \ if using Extended Regular Expressions.

Thus the action of escaping switches between the two modes, and the -E flag is used to enable ERE mode in grep.

When we want to only match the symbols literally, it might thus be better to use BRE, as it is more strict

and less likely to match unintended patterns.

The following table (Table 6.1) shows when to escape the character in which mode.

**Table 6.1:** Differences between BRE and ERE

	Use Literal Symbol	Use ERE Special Syntax
BRE	+	\+
ERE	\+	+

Let us also demonstrate this using an example.

```

1 $ echo -e "a+b\naapple\nbob" > demo.txt
2 $ cat demo.txt
3 a+b
4 aapple
5 bob
6 $ grep 'a+' demo.txt # matches literally
7 a+b
8 $ grep 'a\+' demo.txt # uses special meaning
9 a+b
10 aapple
11 $ grep -E 'a+' demo.txt # uses special meaning
12 a+b
13 aapple
14 $ grep -E 'a\+' demo.txt # matches literally
15 a+b

```

When we use `grep` without the `-E` flag, it uses BRE by default. We have to escape the `+` symbol to use its special meaning. However, when we use the `-E` flag, we can use the `+` symbol directly without escaping it when using as a quantifier. However, if we want to match the symbol literally, we need to escape it in ERE but not in BRE. In this example, when matching the `+` symbol literally, we get only one line of output, which contains the literal symbol `+`. When using `+` as a quantifier, we get both the lines, since it means one or more `a`, and both lines have one or more `a`. The line `bob` is never printed, as it does not contain any `a` characters.

## 6.5 Perl-Compatible Regular Expressions

9: The Portable Operating System Interface is a family of standards specified by the IEEE Computer Society for maintaining compatibility between operating systems. POSIX defines both the system and user-level application programming interfaces (APIs), along with command line shells and utility interfaces, for software compatibility (portability) with variants of Unix and other operating systems. POSIX is also a trademark of the IEEE. POSIX is intended to be used by both application and system developers.

10: Python and Ruby support PCRE through external libraries.

While POSIX has defined the BRE and ERE standards, Perl has its own regular expression engine that is more powerful and flexible. POSIX<sup>9</sup> specifications are meant to be portable amongst different flavors of Unix and other languages, thus most programming languages also support BRE or ERE, or a similar superscript of them.

However, Perl has its own regular expression engine that is more powerful and flexible. Perl-Compatible Regular Expressions (PCRE) is a project written in C inspired by the Perl Regex Engine. Although PCRE originally aimed at feature equivalence with Perl Regex, the two are not fully equivalent. To study the nuanced differences between PRE and PCRE, you can go through the [Wikipedia page](#).

PCRE is way more powerful than ERE, with some additional syntax and features. It is supported by some programming languages, including Perl, PHP, Python<sup>10</sup>, and Ruby. It is also supported by some text processing utilities, like grep, but not by sed, and awk.

Will we not dive deep into PCRE, as it is a vast topic and is not supported by most text processing utilities. However, feel free to explore PCRE online.

Some of the features of PCRE are:

### 6.5.1 Minimal Matching (a.k.a. "ungreedy")

A `?` may be placed after any repetition quantifier to indicate that the shortest match should be used. The default is to attempt the longest match first and backtrack through shorter matches: e.g. `a.*?b` would match first "ab" in "ababab", where `a.*b` would match the entire string.

If the `U` flag is set, then quantifiers are ungreedy (lazy) by default, while `?` makes them greedy.

### 6.5.2 Multiline matching

`^` and `$` can match at the beginning and end of a string only, or at the start and end of each "line" within the string, depending on what options are set.

### 6.5.3 Named subpatterns

A sub-pattern (surrounded by parentheses, like `(...)`) may be named by including a leading `?P<name>` after the opening parenthesis. Named subpatterns are a feature that PCRE adopted from Python regular expressions.

This feature was subsequently adopted by Perl, so now named groups can also be defined using `?<name>...|` or `\l{name}|?name'...)|`, as well as `(?P<name>...).`

Named groups can be backreferenced with, for example: `(?P=name)` (Python syntax) or `\k'<i>name'` (Perl syntax).

### 6.5.4 Look-ahead and look-behind assertions

This is one of the most useful features of PCRE. Patterns may assert that previous text or subsequent text contains a pattern without consuming matched text (zero-width assertion). For example, `/\w+(?=\\t)/` matches a word followed by a tab, without including the tab itself.

Look-behind assertions cannot be of uncertain length though (unlike Perl) each branch can be a different fixed length. `\K` can be used in a pattern to reset the start of the current whole match. This provides a flexible alternative approach to look-behind assertions because the discarded part of the match (the part that precedes `\K`) need not be fixed in length.

The regex matches either the left bound of a word (`<`), the right bound of a word (`>`), the start of line anchor (`^`), or the end of line anchor (`$`).

So, the word boundary match `\b` can be emulated using look-ahead and look-behind assertions: `(?=<\\w) | (?=<=\\w) | (?=\\W) | ^ | $`

---

#### Regex lookahead/lookbehind cheat sheet

<code>(?= ... ) - positive lookahead</code>	<code>(?&lt;= ... ) - positive lookbehind</code>
<code>/\\b(?=\\w)/</code> ↳ "Match '\\b\\b' followed by '\\w'"	<code>/(?&lt;=\\w)\\b/</code> ↳ "Match '\\b\\b' led by '\\w'"
<code>/\\b(?!\\w)/.exec('\\b\\b\\b'); // ✅</code> <code>/\\b(?!\\w)/.exec('\\b\\b\\K'); // null</code>	<code>/(?&lt;=\\w)\\b/.exec('\\b\\b\\b'); // ✅</code> <code>/(?&lt;=\\w)\\b/.exec('\\b\\b\\K'); // null</code>
<code>(?! ... ) - negative lookahead</code>	<code>(?&lt;! ... ) - negative lookbehind</code>
<code>/\\b(?!\\w)/</code> ↳ "Match '\\b\\b' not followed by '\\w'"	<code>/(?&lt;!\\w)\\b/</code> ↳ "Match '\\b\\b' not led by '\\w'"
<code>/\\b(?!\\w)/.exec('\\b\\b\\b'); // ✅</code> <code>/\\b(?!\\w)/.exec('\\b\\b\\K'); // null</code>	<code>/(?&lt;!\\w)\\b/.exec('\\b\\b\\b'); // ✅</code> <code>/(?&lt;!\\w)\\b/.exec('\\b\\b\\K'); // null</code>

---

Figure 6.1: Positive and Negative Look-ahead and look-behind assertions

### 6.5.5 Comments

A comment begins with `(#` and ends at the next closing parenthesis.

### 6.5.6 Recursive patterns

A pattern can refer back to itself recursively or to any subpattern. For example, the pattern `\((a*|(?R))*\)` will match any combination of balanced parentheses and "a"s.

## 6.6 Other Text Processing Tools

Now that we have discussed the basics of regular expressions, let us see how we can use them in some text processing utilities. We will discuss the following text processing utilities:

- ▶ `tr` - Translate characters.
- ▶ `cut` - Cut out fields (columns) from a line.
- ▶ `grep` - Search for patterns in a file.
- ▶ `sed` - Stream editor - search and replace, insert, select, delete, translate.
- ▶ `awk` - A programming language for text processing.

However, these are not all the text processing tools that exist. There are many other text processing utilities that are used in Unix-like systems. Some of them are:

- ▶ `rg` - ripgrep - A search tool that combines the usability of The Silver Searcher (`ag`) with the raw speed of `grep`. Useful to find files recursively in a directory or git repository.
- ▶ `fzf` - Fuzzy Finder - A command-line fuzzy finder. Useful to search for files and directories even when you might not know a valid substring of the text. It works by searching for subsequences instead of substrings and other fuzzy search logic. It is extremely powerful when paired with other applications as an interactive select menu.
- ▶ `csvlens` - A tool to view and query CSV files. It is useful to view and query CSV files in a tabular format. It can search for data in a CSV using regex.
- ▶ `pdfgrep` - A tool to search for text in PDF files. It is useful to search for text in PDF files. It

can search for text in a PDF using regex. It also supports PCRE2.

There are other useful utilities which are not part of GNU coreutils, but are very useful in text processing. Feel free to find such tools, install them, and play around with them. However, we won't be able to discuss those in detail here.

### 6.6.1 tr

`tr` is a command that is used to translate characters. It is used to replace characters in a string with other characters. It is useful when we want to replace a list of characters with another list of characters, or remove a character from a string. It is trivial to create rotation ciphers using `tr`.

Without any flags, `tr` takes two arguments, `LIST1` and `LIST2`, which are the two lists of characters. `LIST1` is the source map and `LIST2` is the destination map of the translation. A character can only map to a single character in a translation, however multiple characters can map to the same character. That is, it can be many-to-one, but not one-to-many.

A simple example of `tr` is to convert a single character to another character.

```
1 $ echo "hello how are you" | tr 'o' 'e'
2 helle hew are yeu
```

However, the true power of `tr` is when we use character lists. We can use character lists to replace multiple characters with other characters.

```
1 $ echo "hello how are you" | tr 'a-z' 'A-Z'
2 HELLO HOW ARE YOU
```

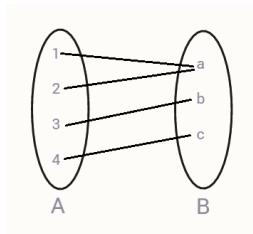


Figure 6.2: Many-to-one mapping

`tr` can also toggle case of characters, that is, the `LIST1` and `LIST2` can have common characters.

```
1 | $ echo "Hello How Are You" | tr 'a-zA-Z' 'A-Za
   -z'
2 | hELLO hOW aRE yOU
```

We do not need to surround the character ranges in brackets.

## Ciphers

**Definition 6.6.1** (Cipher) A cipher is an algorithm for performing encryption or decryption—a series of well-defined steps that can be followed as a procedure. An alternative, less common term is encipherment. To encipher or encode is to convert information into cipher or code. In cryptography, encryption is the process of encoding information. This process converts the original representation of the information, known as plaintext, into an alternative form known as ciphertext. Ideally, only authorized parties can decipher a ciphertext back to plaintext and access the original information.

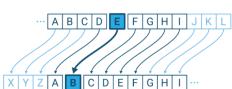


Figure 6.3: Caesar Cipher

11: This is special because the shift of 13 is the same for both encoding and decoding. This is because the English alphabet has 26 characters, and 13 is half of 26. Thus, if we shift by 13, we will get the same character when we shift back by 13. It is thus an involution, that is, applying it twice will give the original text.

One of the common ciphers are rotation ciphers, where each character is replaced by another character that is a fixed number of positions down the alphabet. This is also known as the Caesar cipher, named after Julius Caesar, who is said to have used it to communicate with his generals.

If the shift is 13<sup>11</sup> then the cipher is called ROT13. It is a simple letter substitution cipher that replaces a letter with the 13th letter after it in the alphabet. ROT13 is a special case of the Caesar cipher which was developed in ancient Rome.

Let us try to implement ROT13 using tr.

```

1 $ echo "hello how are you?" | tr 'a-zA-Z' 'n-
    za-mN-ZA-M'
2 uryyb ubj ner lbh?
3 $ echo "uryyb ubj ner lbh?" | tr 'a-zA-Z' 'n-
    za-mN-ZA-M'
4 hello how are you?
```

Observe how running the output of the cipher through the same cipher gives us back the original plain-text. This is because ROT13 is an involution cipher.

We can concatenate multiple ranges of characters in the character-lists as seen above, tr simply converts each character in LIST1 to its corresponding character in LIST2. The length of both the lists should thus be same. However, if the LIST2 is smaller than LIST1, tr will simply repeat the last character of LIST2 as many times as required to make both the lists same length.

```

1 $ echo "abcdefghijklmnopqrstuvwxyz" | tr 'a-z'
    '1'
2 11111111111111111111111111111111
3 $ echo "abcdefghijklmnopqrstuvwxyz" | tr 'a-z'
    '12'
4 12222222222222222222222222222222
5 $ echo "abcdefghijklmnopqrstuvwxyz" | tr 'a-z'
    '123'
6 12333333333333333333333333333333
7 $ echo "abcdefghijklmnopqrstuvwxyz" | tr 'a-z'
    '1234-9'
8 12345678999999999999999999999999
```

The characters in the list are treated as characters, and not digits, thus we cannot replace a character with a multi-digit number. the range 1-26 will not result to 26 numbers from 1 to 26, rather, it results in three numbers: 1, 2, and 6. Similarly 1-72 means

1,2,3,4,5,6,7,2.

```
1 $ echo "abcdefghijklmnopqrstuvwxyz" | tr 'a-z' '1-26'  
2 1266666666666666666666666666  
3 $ tr 'a-z' '1-72' <<< "  
4 abcdefghijklmnopqrstuvwxyz"  
5 12345672222222222222222222
```

We can also repeat a character any arbitrary number of times by using the \* character in the LIST2 inside square brackets.

```
1 $ echo "abcdefghijklmnopqrstuvwxyz" | tr 'a-z' '1-4[5*10]7'  
2 12345555555557777777777777
```

Here the repeat number is actually treated as a number, and thus multi-digit numbers (as shown above) can be used to repeat the character any number of times.

`tr` can perform any cipher that does not depend on additional memory or state.

## Deletion

`tr` can also delete or drop characters from a string of characters. The flag `-d` is used to delete characters from the input string. The syntax is `ls -d 'LIST1'`, where `LIST1` is the list of characters to delete.

```
1 $ echo "hi! hello how are you?" | tr -d 'aeiou'  
2 h! hll hw r y?
```

Here we are deleting all the vowels from the input string.

We can also use ranges to delete characters.

```

1 $ echo "hi! hello how are you?" | tr -d 'a-m'
2 ! o ow r you?

```

Here we are deleting all the characters from a to m.

Sometimes the characters we want to delete is a lot, and its easier to specify the characters we want to keep. We can use the -c flag to complement the character list, that is, to keep the characters that are in the list.

```

1 $ echo "hi! hello how are you?" | tr -cd 'a-m'
2 hihellhae

```

Here we are keeping only the characters from a to m. Observe that it also deletes the punctuations, spaces and the newline characters as well.

This is useful if we want to filter out only some characters from a stream of random characters, for example when trying to generate a random<sup>12</sup> password.

```

1 $ tr -cd 'a-zA-Z0-9' < /dev/urandom | head -c
   20 ; echo
2 8J0zmr4BUbho6wDPaipT

```

This uses the /dev/urandom file to generate random characters, and then filters out only the alphanumeric characters. The head -c 20 is used to print only the first 20 characters, and the echo is used to print a newline after the password.

## Squeeze

Finally, tr can also be used to squeeze characters, that is, to replace multiple consecutive occurrences of a character with a single occurrence. The -s flag is used to squeeze characters. It also takes a

12: It is not recommended to use /dev/random instead of /dev/urandom as it will block if there is a lack of enough entropy. We should totally avoid using RANDOM variable for cryptographic works since it is not cryptographically secure random number. Computers cannot really generate random numbers, since they are deterministic. Most random number generators simply use external variables like the voltage, temperature, and microphone noise to simulate randomness. This may seem random to humans but is not cryptographically secure for use in generating passwords. There are more secure algorithms to generate passwords. Read more about it [here](#).

single argument, which is the list of characters to squeeze.

Use single quotes for this example, and not double quotes, as !! has special meaning in double quotes. It expands to the last run command. This is useful when you write a long command and forget to use sudo, instead of typing the entire thing again, or using arrow keys, simply type sudo !! to expand the !! with the entire previous command.

```
1 $ echo 'Hello!!!!! Using multiple punctuations
2           is not only grammatically incorrect but
3           also obnoxious!' | tr -s '!'
Hello! Using multiple punctuations is not only
grammatically incorrect but also obnoxious
!
```

## 6.6.2 cut

If you want to extract a certain column, or a certain range of columns from a structured text file, doing so using regular expressions can be a bit cumbersome.

```
1 $ cat data.csv
2 name,age,gender
3 Alice,18,F
4 Bob,32,M
5 Carla,23,F
6 $ grep -P '[^,]*,\K[^,]*(?=,[^,]*)' data.csv -
7   o
8 age
9 18
10 32
11 23
```

As seen above, we can use regular expressions to extract the second column from a CSV file. However, this is not very readable, and can be cumbersome for large files with a lot of columns. This also requires using PCRE to avoid matching the lookbehind and the lookaheads. However, this can be done in an easier manner. This is where the `cut` command comes in.

Using `cut`, this operation becomes trivial.

```

1 $ cat data.csv
2 name,age,gender
3 Alice,18,F
4 Bob,32,M
5 Carla,23,F
6 $ cut -d, -f2 data.csv
7 age
8 18
9 32
10 23

```

The `-d` flag is used to specify the delimiter, and the `-f` flag is used to specify the field. The delimiter is the character that separates the fields, and the field is the column that we want to extract. The fields are numbered starting from 1.

`cut` can also extract a range of columns.

```

1 $ cat data.csv
2 name,age,gender
3 Alice,18,F
4 Bob,32,M
5 Carla,23,F
6 $ cut -d, -f1,3 data.csv
7 name,gender
8 Alice,F
9 Bob,M
10 Carla,F
11 $ cut -d, -f2-3 data.csv
12 age,gender
13 18,F
14 32,M
15 23,F
16 $ cut -d, -f2- data.csv
17 age,gender
18 18,F
19 32,M
20 23,F

```

We can also mention disjoint sets of columns or ranges of columns separated by commas.

Lets try to parse the PCRE regex `[^,]*,\K[^,]*(?=,[^,]*)` to extract the second column from a CSV file.

The first part, `[^,]*`, matches the first column, and the `\K` is used to reset the start of the match. This ensures that the first column is present, but not matched and thus not printed.

The second part, `[^,]*` matches the second column. We are matching as many non-comma characters as possible.

The `(?=,[^,]*)` is a lookahead assertion, and is used to match the third column. This ensures that it matches only the second column, and no other column. It will ensure a third column is present, but not match it, thus not printing it.

We had to use `\K` and we could not use a lookbehind assertion, as lookbehind assertions are fixed length, and we do not know the length of the first column. We could have used a lookbehind assertion if we knew the length of the first column.

The range is inclusive, that is, it includes the start and end columns.

If the start of the range is absent, it is assumed to be the first column.

If the end of the range is absent, it is assumed to be the last column.

This lets us extract columns even if we do not know the number of columns in the file.

Here we are using the /etc/passwd file as an example. The /etc/passwd file is a text file that contains information about the users on the system. It contains information like the username, user ID, group ID, home directory, and shell. The file is a colon-separated file, where each line contains information about a single user. The fields are separated by colons, and the fields are the username, password (it is not stored, so it is always x), user ID, group ID, user information (this is usually used by modern distributions to store the user's full name), home directory, and shell. The file is readable by all users, but only writable by the root user. The file is used by the system to authenticate users and to store information about the users on the system.

```
$ cut -d: -f1,5-7 /etc/passwd | tail -n5
dhcpcd:dhcpcd privilege separation:/:/usr/bin/
    nologin
redis:Redis in-memory data structure store:/
    var/lib/redis:/usr/bin/nologin
saned:SANE daemon user:/:/usr/bin/nologin
tor::/var/lib/tor:/usr/bin/nologin
test1::/home/test1:/usr/bin/bash
```

**Exercise 6.6.1** Now that you are familiar with the /etc/passwd file, try to extract the usernames and the home directories of the users. The usernames are the first field, and the home directories are the sixth field. Use the cut command to extract the usernames and the home directories of the users.

## Delimiter

The default delimiter for cut is the tab character. However, we can specify the delimiter using the -d flag. Although it is not required to quote the delimiter, certain characters might be apprehended by the shell and not passed as-is to the command, hence it is always best practice to quote the delimiter using single-quotes. The delimiter has to be a single character, and cannot be more than one character.

The input delimiter can only be specified if splitting the file by fields, that is, when we are using -f flag with a field or range of fields.

## Output Delimiter

The output delimiter by default is the same as the input delimiter. However, we can specify the

output delimiter using the `--output-delimiter` flag. The output delimiter can be a string, and not just a single character. This is useful when we want to change the delimiter of the output file.

```

1 $ head -n1 /etc/passwd | cut -d: -f5- --output
   -delimiter=,
2 root,/root,/usr/bin/bash
3 $ head -n1 /etc/passwd | cut -d: -f5- --output
   -delimiter=,
4 root,,/root,,/usr/bin/bash

```

`cut`, like most coreutils, will read the data from standard input (stdin) if no file is specified. This is useful when we want to pipe the output of another command to `cut`.

## Character Range

The `-b` flag is used to extract bytes from a file. The bytes are numbered starting from 1. The byte range is inclusive, that is, it includes the start and end bytes. If the start of the range is absent, it is assumed to be the first byte. If the end of the range is absent, it is assumed to be the last byte.

If we are working with a file with multi-byte characters, the `-b` flag will not work as expected, as it will extract bytes, and not characters. Then we can use the `-c` flag to extract characters from a file. The characters are numbered starting from 1. However, in GNU `cut`, the feature is not yet implemented. If you are using **freebsd cut** then the difference can be observed.

```

1 $ head -n1 /etc/passwd | cut -b5-10
2 :x:0:0
3 $ head -n1 /etc/passwd | cut -c5-10
4 :x:0:0

```

## Complement

Sometimes its easier to specify the fields we want to drop, rather than the fields we want to keep. We

can use the `--complement` flag to drop the fields we specify.

Recall that the second field of the `/etc/passwd` file is the password field, which is always `x`. We can drop this field using the `--complement` flag.

```

1 $ head -n1 /etc/passwd
2 root:x:0:0:root:/root:/usr/bin/bash
3 $ head -n1 /etc/passwd | cut -d: --complement
   -f2
4 root:0:0:root:/root:/usr/bin/bash

```

## Only Delimited Lines

Sometimes we have a semi-structured file, where some lines are delimited by a character, and some are not. We can use the `--only-delimited` flag to print only the lines that are delimited by the delimiter.

Note that if we mention a field number that is not present in the line, `cut` will simply print nothing. If we print fields 1 to 3, and there are only 2 fields, it will print only first two fields, etc.

```

1 $ cat data.csv
2 name,age,gender
3 Alice,18,F
4 Bob,32,M
5 Carla,23,F
6 # This is a comment
7 $ cut -d, -f1,3 data.csv
8 name,gender
9 Alice,F
10 Bob,M
11 Carla,F
12 # This is a comment
13 $ cut -d, -f1,3 data.csv --only-delimited
14 name,gender
15 Alice,F
16 Bob,M
17 Carla,F

```

`Cut` is a very handy tool for text processing, we will be using it extensively in the upcoming chapters.

### 6.6.3 paste

`paste` is a very simple command that is used to merge lines of files. It is used to merge lines of files horizontally, that is, to merge lines from multiple files into a single line. It is useful when we want to merge lines from multiple files into a single line, and separate them by a delimiter. It can also be used to join one file into a single line separated by a delimiter.

```

1 $ cat file1.txt
2 hello world
3 this is file1
4 $ cat file2.txt
5 this is file2
6 and it has more lines
7 than file1
8 $ paste file1.txt file2.txt
9 hello world      this is file2
10 this is file1   and it has more lines
11                      than file1

```

The default delimiter is the tab character, however we can specify the delimiter using the `-d` flag.

```

1 $ paste -d: file1.txt file2.txt
2 hello world:this is file2
3 this is file1:and it has more lines
4 :than file1

```

`Paste` can also be used to merge lines from a single file into a single line. The `-s` flag is used to merge lines from a single file into a single line. We can specify the delimiter for this as well using the `-d` flag.

```

1 $ paste -d: -s file1.txt
2 hello world:this is file1

```

This is better than using `tr '\n' ':'` to replace the newline character with a delimiter, as `tr` will also

replace the last newline character of the last line, giving a trailing delimiter.

```
1 $ tr '\n' ':' < file1.txt
2 hello world:this is file1:
```

This is very helpful when we want to find the sum of numbers in a file.

```
1 $ cat numbers.txt
2 1
3 4
4 2
5 6
6 7
7 2
8 $ paste -sd+ numbers.txt
9 1+4+2+6+7+2
10 $ paste -sd+ numbers.txt | bc
11 22
```

Here we are first using `paste` to merge the lines of the file into a single line, separated by the `+` character. We then pipe this to `bc`, which is a command line calculator, to calculate the sum of the numbers.

#### 6.6.4 fold

Just like we can use `paste` to merge lines of a file, we can use `fold` to split lines of a file. `fold` is a command that is used to wrap lines of a file. It is used to wrap lines of a file to a specified width.

```
1 $ cat data.txt
2 123456789
3 $ fold -w1 data.txt
4 1
5 2
6 3
7 4
8 5
```

```
9 6
10 7
11 8
12 9
13 $ fold -w2 data.txt
14 12
15 34
16 56
17 78
18 9
```

We can also force `fold` to break lines at spaces only, and not in the middle of a word. However, if it is not possible to maintain the maximum width specified if breaking solely on spaces, it will break on non-spaces as well.

```
1 $ cat text.txt
2 This is a big block of text
3 some of these lines can break easily
4 Whereas_some_are_to_long_to_break
5 $ fold -sw10 text.txt
6 This is a
7 big block
8 of text
9 some of
10 these
11 lines can
12 break
13 easily
14 Whereas_so
15 me_are_to_
16 long_to_br
17 eak
```

This is useful if you want to undo the operation of `tr -d '\n'` performed on lines of equal width.

### 6.6.5 grep

grep is a command that is used to search for patterns in a file. It is used to search for a pattern in a file, and print the lines that match the pattern.

Grep has a lot of flags and features, and is a very powerful tool for searching for patterns in a file. It can search using **BRE**, **ERE**, and even **PCRE**.

We will discuss grep in detail in the next chapter.

### 6.6.6 sed

sed is a stream editor that is used to perform basic text transformations on an input stream. It is used to search and replace, insert, select, delete, and translate text.

Sed is a sysadmin's go to tool for performing quick text transformations on files. It can also perform the changes directly on the file, without needing to write the changes to a new file.

We cover sed in detail in later chapters.

### 6.6.7 awk

awk is a programming language that is used for text processing and data extraction. It is a very powerful tool for text processing, and is used to extract and manipulate data from files.

It has its own programming language, although with very few keywords, and is very easy to learn.

We cover awk in detail in later chapters.

# Grep 7

We have already seen and used grep throughout this chapter while discussing regex. grep is a command that is used to search for patterns in a file. It is used to search for a pattern in a file, and print the lines that match the pattern.

The name grep comes from the **g/re/p** command in the **ed** editor. The **ed** editor is a line editor, and the **g/re/p** command is used to search for a pattern in a file, and print the lines that match the pattern. The grep command is a standalone command that is used to search for a pattern in a file, and print the lines that match the pattern.

grep can use **BRE** or **ERE**, or even **PCRE** if the **-P** flag is used. By default, grep matches using the **BRE** engine.

```
1 $ grep "[aeiou]" -o <<< "hello how are you?"  
2 e  
3 o  
4 o  
5 a  
6 e  
7 o  
8 u
```

## 7.1 Regex Engine

However, grep can also use the **ERE** engine using the **-E** flag. This is useful when we want to use the special characters like **+**, **?**, **(**, **)**, **{**, **}**, and **|** without escaping them.<sup>1</sup>

<sup>1</sup>: There is also an executable called **egrep**, which is the same as **grep -E**. It is provided for compatibility with older Unix systems. It is not recommended to use **egrep**, as it is deprecated and not present in all systems. You should use **grep -E** instead.

```

1 $ grep -E "p+" -o <<< "apple and pineapples"
2 pp
3 p
4 pp

```

Sometimes, it is required to not use regex at all, and simply search for a string. We can use the `-F` flag to search for a fixed string, and not a regex. This will search for any line that has the substring. Any symbol that has special meaning in regex will be treated as a literal character.

```

1 $ grep -F "a+b*c=?"
2 a+b*c=?

```

This mode is useful if you want to match any arbitrary string in a file, and do not know what the string is going to be, thus not allowing you to escape the special characters.

```

1 $ cat data.txt
2 Hello, this is a file
3 with a lot of equations
4 1.  $a^2 + b^2 = c^2$  for right angle triangles
5 2.  $E = mc^2$ 
6 3.  $F = ma$ 
7 4. The meaning of life, the universe, and
       everything = 42
8 $ read -r -p "What to search for? " pattern
9 What to search for? ^2
10 $ grep "$pattern" data.txt
11 2.  $E = mc^2$ 
12 $ grep -F "$pattern" data.txt
13 1.  $a^2 + b^2 = c^2$  for right angle triangles
14 2.  $E = mc^2$ 

```

In the above example, you can see a file full of equations, thus special symbols. If we want to dynamically input what to search for, we can use the `read` command to read the input from the user, and then use `grep` to search for the pattern. If we use

the `-F` flag, we can search for the pattern as a fixed string, and not as a regex.

Here we wanted to find all the quadratic equations, and thus searched for `^2`. However, observe that the grep, without the `-F` flag, did not match the first equation which has multiple `^2` in it, because it is treating `^` with its special meaning of **start of line anchor**.

If we were statically providing the search string, we can simply escape the special characters, and use grep without the `-F` flag. But in cases like this, it is easier to simply use the `-F` to not use regular expression and simply find substrings.

## 7.2 PCRE

Similarly, there are situations where the ERE engine is not powerful enough and we need to use the PCRE engine. We can use the `-P` flag to use the PCRE engine.

```

1 $ cat data.txt
2 Hello, this is a file
3 with a lot of equations
4 1. a^2 + b^2 = c^2 for right angle triangles
5 2. E = mc^2
6 3. F = ma
7 4. The meaning of life, the universe, and
       everything = 42
8 $ grep -P "c\^2" data.txt
9 1. a^2 + b^2 = c^2 for right angle triangles
10 2. E = mc^2
11 $ grep -P "c\^2(?=.*triangle)" data.txt
12 1. a^2 + b^2 = c^2 for right angle triangles

```

Here, if we want to find all the equations with `c^2`, but only if that equation also mentions "triangle"

somewhere after the `c^2`, then we can use lookahead assertions of PCRE to accomplish that. Here, the `.*` matches any character zero or more times, and the `triangle` matches the string "triangle". Putting it inside a lookahead assertion `((?= ))` ensures that the pattern is present, but not part of the match.

This can be confirmed by using the `-o` flag to print only the matched part of the line.

```
1 | $ grep -P "c^2(?=.*triangle)" -o data.txt
2 | c^2
```

## 7.3 Print Only Matching Part

We have been using the `-o` flag extensively to print only the matching part of the line. This is useful when we want to extract only the part of the line that matches the pattern and not the entire line. This is also very useful for debugging regex, as we can see what part of the line is actually matching the pattern.

If we do not use `-o`, then any line having one or more match will be printed entirely, and the matches will be colored red<sup>2</sup>, however, if two consecutive matches are present, it becomes hard to distinguish between the two matches.

If we use the `-o` flag, then only the matching part of the line is printed, and each match is printed on a new line, making it easy to see exactly which parts are matched.

2: grep will color the match only if you pass the flag `--color=always`, or if the flag is set to `--color=auto` and the terminal supports color output. Otherwise no ANSI-escape code is printed by grep.

You can also change the color of the match by setting the `GREP_COLORS` environment variable. The default color is red, but you can change it to any color you want.

```
1 | $ grep -E "o{,3}" <<< "hellooooo"
2 | hellooooo
3 | $ grep -Eo "o{,3}" <<< "hellooooo"
4 | ooo
5 | oo
```

In the above example, we ask grep to match the pattern `o{,3}`, that is, match the letter `o` zero to three times. If we do not use the `-o` flag, then the entire line is printed, and the matches are colored red. This however creates a confusion, even though all the 5 `o`'s are matched, they cannot obviously be a single match, since a single match can only be of a maximum of 3 `o`'s. So how are the matches grouped? Is it 3 `o`'s followed by 1 `o` followed by another `o`? Is it 2 `o`'s followed by 2 `o`'s followed by one `o`? There seems to be no way to tell from the first output.

However, if we use the `-o` flag, then only the matching part of the line is printed, and each match is printed on a new line. It then becomes clear that grep will greedily match as much as possible first, so the first three `o`'s are matched, and then the remaining are grouped as a single match.

## 7.4 Matching Multiple Patterns

### 7.4.1 Disjunction

If we want to match multiple patterns, we can use the `-e` flag to specify multiple patterns. This is useful when we want to match multiple patterns, and not just a single pattern. Any line containing one or more of the patterns we are searching for will be printed. This is like using an **OR** clause.

```

1 $ cat data.txt
2 Hello, this is a file
3 with a lot of equations
4 1. a^2 + b^2 = c^2 for right angle triangles
5 2. E = mc^2
6 3. F = ma

```

In this example, we want to find lines that match the word "file" or the word "life". We can use the `-e` flag to specify multiple patterns. The `-e` flag is automatically implied if we are searching for a single pattern, however, if we are searching for more than one pattern, we have to specify it for all of the patterns, including the first one.

```

7 4. The meaning of life, the universe, and
   everything = 42
8 $ grep "file" data.txt
9 Hello, this is a file
10 $ grep -e "file" -e "life" data.txt
11 Hello, this is a file
12 4. The meaning of life, the universe, and
   everything = 42

```

## 7.4.2 Conjunction

If we want to match lines that contain all of the patterns we are searching for, we can pipe the output of one grep to another, to do iterative filtering.

In this example, first we show each of the patterns we are matching, and that they output multiple lines. Then finally we combine both the patterns using pipe to find only the common line in both the outputs. The patterns can be specified in either order. We need to provide the file only in the first grep, and we should not provide a file name to any of the other greps, otherwise it will not use the output of the previous grep as input and just filter from the file instead.

```

1 $ cat data.txt
2 Hello, this is a file
3 with a lot of equations
4 1. a^2 + b^2 = c^2 for right angle triangles
5 2. E = mc^2
6 3. F = ma
7 4. The meaning of life, the universe, and
   everything = 42
8 $ grep '\ba\b' data.txt
9 Hello, this is a file
10 with a lot of equations
11 1. a^2 + b^2 = c^2 for right angle triangles
12 $ grep '^[0-9]' data.txt
13 1. a^2 + b^2 = c^2 for right angle triangles
14 2. E = mc^2
15 3. F = ma
16 4. The meaning of life, the universe, and
   everything = 42
17 $ grep '\ba\b' data.txt | grep '[0-9]'
18 1. a^2 + b^2 = c^2 for right angle triangles

```

Here the regex `^[0-9]` matches any line that starts with a number. And the regex `\ba\b` matches any line that contains the word "a". The \b is a word boundary, and matches the start or end of a word. Thus it will match the word "a" and not the letter "a" in a word.

## 7.5 Read Patterns from File

If we have a lot of patterns to search for, we can put them in a file, and use the `-f` flag to read the patterns from the file. Each line of the file is treated as a separate pattern, and any line that matches any of the patterns will be printed. The type of regex engine used depends on whether `-E`, `-P`, or `-F` is used.

```
1 $ cat data.txt
2 p+q=r
3 apple
4 e*f=g
5 $ cat pattern
6 p+
7 e*
8 $ grep -G -f pattern data.txt -o
9 p+
10 e
11 e
12 $ grep -F -f pattern data.txt -o
13 p+
14 e*
15 $ grep -E -f pattern data.txt -o
16 p
17 pp
18 e
19 e
```

## 7.6 Ignore Case

If we want to ignore the case of the pattern, we can use the `-i` flag. This is useful when we want to match a pattern, but do not care about the case of the pattern, or we are not sure what the case is.

```
1 $ grep 'apple' /usr/share/dict/words | head
```

```
2 appleberry
3 appleblossom
4 applecart
5 apple-cheeked
6 appled
7 appledane
8 appledrone
9 apple-eating
10 apple-faced
11 apple-fallow
12 $ grep -i 'apple' /usr/share/dict/words | head
13 Apple
14 appleberry
15 Appleby
16 appleblossom
17 applecart
18 apple-cheeked
19 appled
20 Appledorf
21 appledane
22 appledrone
```

As seen above, the first `grep` matches only the lines that contain the word "apple", and not "Apple". However, the second `grep` matches both "apple" and "Apple".

## 7.7 Invert Match

Sometimes its easier to specify the patterns we do not want to match, rather than the patterns we want to match. We can use the `-v` flag to invert the match, that is, to print only the lines that do not match the pattern.

```
1 $ cat data.txt
2 apple
3 banana
4 blueberry
```

```

5 blackberry
6 raspberry
7 strawberry
8 $ grep -v 'berry' data.txt
9 apple
10 banana

```

This is useful when we want to filter out some arbitrary stream of data for some patterns.

```

1 $ grep -v 'nologin$' /etc/passwd
2 root:x:0:0:root:/root:/usr/bin/bash
3 git:x:971:971:git daemon user:/:/usr/bin/git-
    shell
4 ntp:x:87:87:Network Time Protocol:/var/lib/ntp
    :/bin/false
5 sayan:x:1000:1001:Sayan:/home/sayan:/bin/bash
6 test1:x:1001:1002::/home/test1:/usr/bin/bash

```

## 7.8 Anchoring

If we want to match a pattern only at the start of the line, or at the end of the line, we can use the ^ and \$ anchors respectively.

However, in grep, if we want to match a pattern that is the entire line, we can use the -x flag. This is useful when we want to match the entire line, and not just a part of the line. This is same as wrapping the entire pattern in ^\$, but is more readable.

Similarly, if we want to match a pattern that is a word, we can use the -w flag. This is useful when we want to match a word, and not a part of a word. This is same as wrapping the entire pattern in \\b, but is more readable.

```

1 $ grep 'apple' /usr/share/dict/words | tail
2 stapple
3 star-apple

```

In this example, we are filtering out all the users that have the shell set to nologin. These are users which cannot be logged into. We can use the -v flag to invert the match, and print only the users that do not have the shell set to nologin. Effectively printing all the accounts in the current system that can be logged into. ntp uses the /bin/false shell, which is used to prevent the user from logging in, but is not the same as /usr/bin/nologin, which is used to prevent the user from logging in and also prints a message to the user.

Observe in this example, if we do not use the -w flag then words that have the substring "apple" will also be matched. However, when we use the -w flag, only the word "apple" is matched as a whole word.

```

4 strapple
5 thorn-apple
6 thrapple
7 toffee-apple
8 undappled
9 ungrapple
10 ungrappled
11 ungrappler
12 $ grep -w 'apple' /usr/share/dict/words | tail
13 may-apple
14 oak-apple
15 pine-apple
16 pond-apple
17 rose-apple
18 snap-apple
19 sorb-apple
20 star-apple
21 thorn-apple
22 toffee-apple

```

## 7.9 Counting Matches

Sometimes all we want is to see how many lines match the pattern, and not the lines themselves. We can use the `-c` flag to count the number of lines that match the pattern. This is exactly same as piping the output of `grep` to `wc -l`, but is more readable.

```

1 $ grep -c 'apple' /usr/share/dict/words
2 101
3 $ grep -ic 'apple' /usr/share/dict/words
4 107

```

From this we can quickly see that there are  $107 - 101 = 6$  lines that contain the word "Apple" in the file.

We can also print those lines using the `diff` command.

```

1 $ diff <(grep apple /usr/share/dict/words) <(
    grep -i apple /usr/share/dict/words)
2 0a1
3 > Apple
4 1a3
5 > Appleby
6 5a8
7 > Appledorf
8 10a14
9 > Applegate
10 28a33
11 > Appleseed
12 31a37
13 > Appleton

```

Or by using the `comm` command.

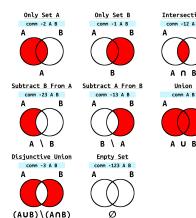
The `comm` command is used to compare two sorted files line by line. It is used to find lines that are common, or different between two files. The `comm` command requires the input files to be **sorted**, and it will not work if the files are not sorted. The `comm` command has three numbered columns, the first column is the lines that are unique to the first file, the second column is the lines that are unique to the second file, and the third column is the lines that are common to both files. The `comm` command is useful when we want to compare two files, and find the differences between them, or find the lines common between them.

```

1 $ comm -13 <(grep apple /usr/share/dict/words
    | sort) <(grep -i apple /usr/share/dict/
    words | sort)
2 Apple
3 Appleby
4 Appledorf
5 Applegate
6 Appleseed
7 Appleton

```

**The 'comm' Command**



**Figure 7.1:** The `comm` command

Observe that we had to sort the files before using `comm`, as `comm` requires the files to be sorted. We can use the `<(command)` syntax to pass the output of a command as a file to another command. This is called process substitution.

## 7.10 Print Filename

Sometimes, we may want to search for a pattern in multiple files, and we want to know which file contains the pattern. We can use the `-H` flag to print the filename along with the matched line. This is however the default behaviour in GNU grep if multiple files are passed.

In this example, we are passing all the `.txt` files in the current directory to `grep`, and searching for patterns. The `-H` flag is implicit and is used to print the filename along with the matched line. This is useful when we are searching for a pattern in multiple files, and we want to know which file contains the pattern.

```

1 $ cat hello.txt
2 hello world
3 hello universe
4 $ cat linux.txt
5 this is linux
6 $ grep hello *txt
7 hello.txt:hello world
8 hello.txt:hello universe
9 $ grep this *txt
10 linux.txt:this is linux
11 $ grep i *txt
12 hello.txt:hello universe
13 linux.txt:this is linux

```

But if we want to suppress the printing of the filename, we can use the `-h` flag. This is useful when we are searching for a pattern in multiple files, and we do not want to know which file contains the pattern, just the line.

```

1 $ cat hello.txt
2 hello world
3 hello universe
4 $ cat linux.txt
5 this is linux
6 $ grep -h hello *txt
7 hello world
8 hello universe
9 $ grep -h this *txt
10 this is linux
11 $ grep -h i *txt
12 hello universe

```

```
13 | this is linux
```

Similarly, if we want to print the filename only if there are multiple files, we can use the `-l` flag. This will print **only** the name of the file that has one or more matches. This mode does not print the filename multiple time even if it has multiple matches on same or different lines. Thus this is not same as `grep pattern files... | cut -d: -f1`. Rather it is same as `grep pattern files... | cut -d: -f1 | uniq`

## 7.11 Limiting Output

Although we can use `head` and `tail` to limit the number of lines of output of `grep`, `grep` also has the `-m` flag to limit the number of matches. This is useful when we want to see only the first few matches, and not all the matches.

```
1 $ grep 'nologin' /etc/passwd
2 bin:x:1:1:::/usr/bin/nologin
3 daemon:x:2:2:::/usr/bin/nologin
4 mail:x:8:12::/var/spool/mail:/usr/bin/nologin
5 ftp:x:14:11::/srv/ftp:/usr/bin/nologin
6 http:x:33:33::/srv/http:/usr/bin/nologin
7 nobody:x:65534:65534:Kernel Overflow User::/
     usr/bin/nologin
8 dbus:x:81:81:System Message Bus:::/usr/bin/
     nologin
9 systemd-coredump:x:984:984:systemd Core Dumper
     ::/usr/bin/nologin
10 systemd-network:x:982:982:systemd Network
      Management:::/usr/bin/nologin
11 systemd-oom:x:981:981:systemd Userspace OOM
      Killer:::/usr/bin/nologin
12 systemd-journal-remote:x:980:980:systemd
      Journal Remote:::/usr/bin/nologin
```

The benefit of using `-m` instead of `head` is that the output will remain colored if coloring is supported, although it is not visible here, try running both to observe the difference.

```
13 systemd-journal-upload:x:979:979:systemd
    Journal Upload:/:/usr/bin/nologin
14 systemd-resolve:x:978:978:systemd Resolver:/::
    usr/bin/nologin
15 systemd-timesync:x:977:977:systemd Time
    Synchronization:/:/usr/bin/nologin
16 tss:x:976:976:tss user for tpm2:/:/usr/bin/
    nologin
17 uuidd:x:68:68:::/usr/bin/nologin
18 avahi:x:974:974:Avahi mDNS/DNS-SD daemon:/::
    usr/bin/nologin
19 named:x:40:40:BIND DNS Server:/:/usr/bin/
    nologin
20 dnsmasq:x:973:973:dnsmasq daemon:/:/usr/bin/
    nologin
21 geoclue:x:972:972:Geoinformation service:/var/
    lib/geoclue:/usr/bin/nologin
22 _talkd:x:970:970:User for legacy talkd server
    ::/:/usr/bin/nologin
23 nbd:x:969:969:Network Block Device:/var/empty
    ::/usr/bin/nologin
24 nm-openconnect:x:968:968:NetworkManager
    OpenConnect:/:/usr/bin/nologin
25 nm-openvpn:x:967:967:NetworkManager OpenVPN
    ::/usr/bin/nologin
26 nvidia-persistenced:x:143:143:NVIDIA
    Persistence Daemon:/:/usr/bin/nologin
27 openvpn:x:965:965:OpenVPN:/:/usr/bin/nologin
28 partimag:x:110:110:Partimage user:/:/usr/bin/
    nologin
29 polkitd:x:102:102:PolicyKit daemon:/:/usr/bin/
    nologin
30 rpc:x:32:32:Rpcbind Daemon:/var/lib/rpcbind:/_
    usr/bin/nologin
31 rpcuser:x:34:34:RPC Service User:/var/lib/nfs
    ::/usr/bin/nologin
32 rtkit:x:133:133:RealtimeKit:/proc:/usr/bin/
    nologin
33 sddm:x:964:964:SDDM Greeter Account:/var/lib/
    sddm:/usr/bin/nologin
```

```
34 usbmux:x:140:140:usbmux user:/:/usr/bin/
      nologin
35 qemu:x:962:962:QEMU user:/:/usr/bin/nologin
36 cups:x:209:209:cups helper user:/:/usr/bin/
      nologin
37 dhcpcd:x:959:959:dhcpcd privilege separation
      :/:/usr/bin/nologin
38 redis:x:958:958:Redis in-memory data structure
      store:/var/lib/redis:/usr/bin/nologin
39 saned:x:957:957:SANE daemon user:/:/usr/bin/
      nologin
40 tor:x:43:43::/var/lib/tor:/usr/bin/nologin
41 $ grep 'nologin' /etc/passwd | head -n5
42 bin:x:1:1::/:/usr/bin/nologin
43 daemon:x:2:2::/:/usr/bin/nologin
44 mail:x:8:12::/var/spool/mail:/usr/bin/nologin
45 ftp:x:14:11::/srv/ftp:/usr/bin/nologin
46 http:x:33:33::/srv/http:/usr/bin/nologin
47 $ grep 'nologin' /etc/passwd -m5
48 bin:x:1:1::/:/usr/bin/nologin
49 daemon:x:2:2::/:/usr/bin/nologin
50 mail:x:8:12::/var/spool/mail:/usr/bin/nologin
51 ftp:x:14:11::/srv/ftp:/usr/bin/nologin
52 http:x:33:33::/srv/http:/usr/bin/nologin

1 $ cat hello.txt
2 hello world
3 hello universe
4 $ cat linux.txt
5 this is linux
6 $ grep -l hello *txt
7 hello.txt
8 $ grep -l this *txt
9 linux.txt
10 $ grep -l i *txt
11 hello.txt
12 linux.txt
```

## 7.12 Quiet Quitting

No, we are not talking about the recent trend of doing the bare minimum in a job.

If we want to suppress the output of grep, and only see if the pattern was found or not, we can use the `-q` flag. This is useful when we want to use grep in a script, and we do not want to see the output of grep, just the exit status. This also implies `-m1`, as we can determine the exit status of grep as soon as we find the first match.

In this example we use concepts such as `if` and `then` to check the exit status of grep. If the exit status is 0, then the pattern was found, and we print "world mentioned". If the exit status is 1, then the pattern was not found, and we do not print anything. `$?` is a special variable that stores the exit status of the last command. If the exit status is 0, then the command was successful, and if the exit status is 1, then the command failed.

We will cover these in later chapters.

```

1 $ cat hello.txt
2 hello world
3 hello universe
4 $ grep -q 'world' hello.txt
5 $ echo $?
6 0
7 $ if grep -q 'world' hello.txt ; then echo "
8     world mentioned" ; fi
9 world mentioned
$ if grep -q 'galaxy' hello.txt ; then echo "
10    galaxy mentioned" ; fi

```

## 7.13 Numbering Lines

If we want to number the lines that match the pattern, we can use the `-n` flag. This is useful when we want to see the line number of the lines that match the pattern.

```

1 $ cat hello.txt
2 hello world
3 hello universe
4 $ grep -n 'hello' hello.txt
5 1:hello world
6 2:hello universe

```

## 7.14 Recursive Search

grep can also search for patterns in directories and subdirectories recursively. We can use the `-r` flag to achieve this. This is useful when we want to search for a pattern in multiple files, and we do not know which files contain the pattern, or if the files are nested deeply inside directories.

By default it starts searching from the current directory, but we can specify the directory to start searching from as the second argument.

```
1 $ mkdir -p path/to/some/deeply/nested/folders/
   like/this
2 mkdir: created directory 'path'
3 mkdir: created directory 'path/to'
4 mkdir: created directory 'path/to/some'
5 mkdir: created directory 'path/to/some/deeply'
6 mkdir: created directory 'path/to/some/deeply/
   nested'
7 mkdir: created directory 'path/to/some/deeply/
   nested/folders'
8 mkdir: created directory 'path/to/some/deeply/
   nested/folders/like'
9 mkdir: created directory 'path/to/some/deeply/
   nested/folders/like/this'
10 $ echo hello > path/to/some/deeply/nested/
    folders/like/this/hello.txt
11 $ echo "hello world" > path/world.txt
12 $ grep -r hello
13 path/world.txt:hello world
14 path/to/some/deeply/nested/folders/like/this/
    hello.txt:hello
15 $ grep -r hello path/to/
16 path/to/some/deeply/nested/folders/like/this/
    hello.txt:hello
```

## 7.15 Context Line Control

3: The **-A** flag is for printing lines **after** the match, the **-B** flag is for printing lines **before** the match, and the **-C** flag is for printing lines both **before** and **after** the match.

```

1 $ grep -n sayan /etc/passwd
2 37:sayan:x:1000:1001:Sayan:/home/sayan:/bin/
   bash
3 $ grep -n sayan /etc/passwd -A2
4 37:sayan:x:1000:1001:Sayan:/home/sayan:/bin/
   bash
5 38-qemu:x:962:962:QEMU user:/:/usr/bin/nologin
6 39-cups:x:209:209:cups helper user:/:/usr/bin/
   nologin
7 $ grep -n sayan /etc/passwd -B2
8 35-sddm:x:964:964:SDDM Greeter Account:/var/
   lib/sddm:/usr/bin/nologin
9 36-usbmux:x:140:140:usbmux user:/:/usr/bin/
   nologin
10 37:sayan:x:1000:1001:Sayan:/home/sayan:/bin/
    bash
11 $ grep -n sayan /etc/passwd -C2
12 35-sddm:x:964:964:SDDM Greeter Account:/var/
   lib/sddm:/usr/bin/nologin
13 36-usbmux:x:140:140:usbmux user:/:/usr/bin/
   nologin
14 37:sayan:x:1000:1001:Sayan:/home/sayan:/bin/
   bash
15 38-qemu:x:962:962:QEMU user:/:/usr/bin/nologin
16 39-cups:x:209:209:cups helper user:/:/usr/bin/
   nologin

```

These were some of the flags that can be used with grep to control the output. There are many more flags that can be used with grep, and you can see them by running `man grep`.

Now that we have learnt some of the flags on their

own, let us try to accomplish certain tasks using multiple flags together.

## 7.16 Finding Lines Common in Two Files

If we have two files which has some lines of text, and our job is to find the lines that are common in both, we can use `comm` if the files are sorted, or we can sort the files before passing it to `comm` if we are allowed to have the output sorted.

But if we are not allowed to sort the files, we can use `grep` to accomplish this.

```
1 $ cat file1.txt
2 this is file1
3 this.*
4 a common line
5 is to check if it misinterprets regex
6 apple
7 $ cat file2.txt
8 this is file2
9 this.*
10 a common line
11 is to check if we are checking fixed strings
12 pineapple
```

Ideally the lines that should be detected as being common are the lines

```
1 this.*
2 a common line
```

Remember that `grep` allows using a file as the pattern, and it will match any line that matches any of the patterns in the file. Let us try to use it to provide one file as a pattern, and the other file as the input.

```

1 | $ grep -f file2.txt file1.txt
2 | this is file1
3 | this.*
4 | a common line

```

Observe that it is also matching the line `this is file1`, which is not common in both the files. This is because the `.*` in the pattern `this.*` is being interpreted as a regex, and not as a literal character. We can use the `-F` flag to treat the pattern as a fixed string, and not as a regex.

```

1 | $ grep -Ff file2.txt file1.txt
2 | this.*
3 | a common line

```

So is that it? It looks like we are getting the correct output. Not yet. If we reverse the order of files, we see that we are not getting the correct output.

```

1 | $ grep -Ff file1.txt file2.txt
2 | this.*
3 | a common line
4 | pineapple

```

This is because the `apple` in `file1.txt` is matching the word `pineapple` in `file2.txt` as a substring.

If we want to print only lines that match the entire line, we can use the `-x` flag.

```

1 | $ grep -Fxf file1.txt file2.txt
2 | this.*
3 | a common line

```

Now we are getting the correct output, in the original file order.

If we were to use `comm`, we would have to sort the files first, and then use `comm` to find the common lines.

```

1 | $ comm -12 file1.txt  file2.txt

```

```
2 comm: file 1 is not in sorted order
3 comm: file 2 is not in sorted order
4 comm: input is not in sorted order
5 $ comm -12 <(sort file1.txt) <(sort file2.
     txt)
6 a common line
7 this.*
```

Observe how the order of output is different.

The trick to mastering grep is to understand the flags, and how they can be combined to accomplish the task at hand. The more you practice, the more you will understand how to use grep effectively. Refer to the practice questions in the VM.



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.

## 8.1 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 8.1.1** (Environment Variables) An environment variable is a variable that is accessible 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 8.1.2 (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.

```
1 | $ 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.

```
1 | $ 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.

```
1 | $ var="Hello World"  
2 | $ export var
```

## 8.2 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.

```
1 $ var="Hello World"
2 $ echo $var
3 Hello World
```

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

```
1 $ date="2024_07_30"
2 $ time="08:30:00"
3 $ echo "$date_$time"
4 08:30:00
5 $ echo "${date}_${time}"
6 2024_07_30_08:30:00
```

Here we want to concatenate the values of the variables `date` and `time` with an underscore in between. However, the shell thinks 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.

**Remark 8.2.1** 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.

```
1 $ echo '$USER is' "$USER"
2 $USER is sayan
3 $ echo "\$HOSTNAME is $HOSTNAME"
4 $HOSTNAME is rex
```

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

### 8.2.1 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.

```
1 $ man echo | sed '/^$/d' | head -n15
2 ECHO(1)

          User Commands

          ECHO(1)

3 NAME
4   echo - display a line of text
5 SYNOPSIS
6     echo [SHORT-OPTION]... [STRING]...
7     echo LONG-OPTION
8 DESCRIPTION
9     Echo the STRING(s) to standard output.
10    -n      do not output the trailing
           newline
```

```
11      -e      enable interpretation of
12      backslash escapes
13      -E      disable interpretation of
14      backslash escapes (default)
15      --help  display this help and exit
16      --version
17          output version information and
18          exit
19
20      If -e is in effect, the following
21      sequences are recognized:
22
23 $ help echo | head -n20 | sed '/^ +$/d'
24 echo: echo [-neE] [arg ...]
25      Write arguments to the standard output.
26      Display the ARGs, separated by a single
27      space character and followed by a
28      newline, on the standard output.
29      Options:
30          -n      do not append a newline
31          -e      enable interpretation of the
32      following backslash escapes
33          -E      explicitly suppress interpretation
34          of backslash escapes
35
36      'echo' interprets the following backslash-
37      escaped characters:
38          \a      alert (bell)
39          \b      backspace
40          \c      suppress further output
41          \e      escape character
42          \E      escape character
43          \f      form feed
44          \n      new line
45          \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.

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.

```

1 $ type -a echo
2 echo is a shell builtin
3 echo is /sbin/echo
4 echo is /bin/echo
5 echo is /usr/bin/echo
6 echo is /usr/sbin/echo

1 $ echo --version
2 --version

1 $ /bin/echo --version
2 echo (GNU coreutils) 9.5
3 Copyright (C) 2024 Free Software Foundation,
   Inc.
4 License GPLv3+: GNU GPL version 3 or later <
   https://gnu.org/licenses/gpl.html>.
5 This is free software: you are free to change
   and redistribute it.
6 There is NO WARRANTY, to the extent permitted
   by law.

7
8 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.

```

1 'echo' interprets the following backslash-
   escaped characters:
2      \a      alert (bell)

```

```

3   \b      backspace
4   \c      suppress further output
5   \e      escape character
6   \E      escape character
7   \f      form feed
8   \n      new line
9   \r      carriage return
10  \t      horizontal tab
11  \v      vertical tab
12  \\      backslash
13  \0nnn    the character whose ASCII code
14    is NNN (octal). NNN can be
15    0 to 3 octal digits
16  \xHH     the eight-bit character whose
17    value is HH (hexadecimal). HH
18    can be one or two hex digits
19  \uHHHH    the Unicode character whose
20    value is the hexadecimal value HHHH.
21    HHHH can be one to four hex
22    digits.
23    \UHHHHHHHHH the Unicode character whose
24    value is the hexadecimal value
25    HHHHHHHHHH. HHHHHHHHHH can be one
26    to eight hex digits.

```

```

1 $ echo -e "abc\bde"
2 abd
1 $ echo -e "abc\cde"
2 abc
1 $ echo -e "abc\ede"
2 abce
1 $ echo -e "abc\fde"
2 abc
3   de
1 $ echo -e "abc\vde"
2 abc
3   de
1 $ echo -e "abc\rde"

```

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

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

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.

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.

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`.

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

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.

## 8.2.2 Accessing and Updating Numeric Variables

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.

If the variable is a number, we can perform arithmetic operations on it in 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.

```
1 $ var=5
2 $ echo $var
3 5
4 $ echo $((var+5))
5 10
6 $ echo $((var-5))
7 0
8 $ echo $((var*5))
```

```

9 25
10 $ echo $((var/5))
11 1
12 $ echo $((var%5))
13 0
14 $ echo $((var**2))
15 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.

```

1 $ echo $((var^2))
2 7
3 $ echo $((var&3))
4 1
5 $ echo $((var|3))
6 7
7 $ echo $((~var))
8 -6

```

## Comparison Operations

The comparison operators return 1 if the condition is true, and 0 if the condition is false. This is the

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](#).

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 environment actually exits with a zero exit code if the value is non-zero, fixing this issue.

```

1 $ echo $((var>5))
2 0
3 $ echo $((var>=5))
4 1
5 $ echo $((var<5))
6 0
7 $ echo $((var<=5))
8 1
9 $ echo $((var==5))
10 1
11 $ echo $((var!=5))
12 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.

```

1 $ echo $((++var))
2 6
3 $ echo $((var++))
4 6
5 $ echo $var
6 7
7 $ echo $((--var))
8 6
9 $ echo $((var--))
10 6
11 $ echo $var
12 5

```

## Assignment and Multiple Clauses

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

```

1 $ echo $((a=5, b=10, a+b))
2 15
3 $ echo $((a=5, b=10, a+b, a*b))
4 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.

```

1 $ echo $((a=5, b=10))
2 10
3 $ echo $((a=b=7, a*b))
4 49

```

## Floating Point Arithmetic

Bash does not support floating point arithmetic.

```

1 $ echo $((10/3))
2 3
3 $ echo $((40/71))
4 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.

```

1 $ printf "%.2f\n" "$((10**4 * 10/3))e-4"
2 3.33
3 $ printf "%.2f%\n" "$((10**4 * 40/71))e-4"
4 56.33%

```

```

1 $ awk 'BEGIN { printf "%.2f%", (40/71*100) }'
2 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:

```
1 | base#number
```

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

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

```

1 $ echo $((2#1000 + 2#101))
2 13
3 $ echo $((8#52 + 8#21))
4 59
5 $ echo $((052 + 021))
6 59
7 $ echo $((16#1a + 16#2a))
8 68
9 $ echo $((0x1a + 0x2a))
10 68
11 $ echo $((8#31 - 25))
12 0
```

We can also mix bases in the same expression.

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

```

1 $ var=5
2 $ ((var++))
3 $ echo $var
4 6
```

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

```

1 $ var=5
2 $ expr $var + 5
3 10
4 $ expr $var \* 5
5 25

1 $ a=5
2 $ echo $a
3 5
4 $ let a++
5 $ echo $a
6 6

```

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.

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.

## 8.3 Removing Variables

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

```

1 $ var="Hello World"
2 $ echo $var
3 Hello World
4 $ unset var
5 $ echo $var

```

This can also be used to remove multiple variables.

```

1 $ var1="Hello"
2 $ var2="World"
3 $ echo $var1 $var2
4 Hello World
5 $ unset var1 var2
6 $ echo $var1 $var2

```

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

```

1 $ var="Hello World"

```

```

2 $ echo $var
3 Hello World
4 $ var=""
5 $ 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.

```

1 $ var="Hello World"
2 $ echo $var
3 Hello World
4 $ test -v var ; echo $?
5 0
6 $ var=""
7 $ echo $var
8
9 $ test -v var ; echo $?
10 0
11 $ unset var
12 $ echo $var
13
14 $ test -v var ; echo $?
15 1

```

## 8.4 Listing Variables

### 8.4.1 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.

```

1 $ set | head -n120 | shuf | head
2 BASH=/bin/bash
3 LC_NUMERIC=en_US.UTF-8
4 LESS_TERMCAP_so=$'\E[01;44;33m'
5 BASH_REMATCH=()
6 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
7 HOSTTYPE=x86_64
8 COLUMNS=187
9 SHELL=/bin/bash
10 BROWSER=thorium-browser

```

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

```

1 $ var1=5
2 $ set | grep var1
3 var1=5

```

### 8.4.2 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.

```

1 $ declare | head -n10
2 ANDROID_HOME=/home/sayan/.android/sdk
3 ANDROID_SDK_ROOT=/home/sayan/.android/sdk
4 AWT_TOOLKIT=MToolkit
5 BASH=/bin/bash
6 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
7 BASH_ALIASES=()
8 BASH_ARGC=( [0]="0" )
9 BASH_ARGV=()
10 BASH_CMDS=()
11 BASH_COMPLETION_VERSINFO=( [0]=“2” [1]=“11”)

```

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

```

1 $ var1=5
2 $ declare | grep var1
3 var1=5

```

### 8.4.3 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.

```

1 $ env | head -n10
2 SHELL=/bin/bash

```

```

3 WINDOWID=100663324
4 COLORTERM=truecolor
5 LANGUAGE=
6 LC_ADDRESS=en_US.UTF-8
7 JAVA_HOME=/usr/lib/jvm/default
8 LC_NAME=en_US.UTF-8
9 SSH_AUTH_SOCK=/run/user/1000/ssh-agent.socket
10 SHELL_SESSION_ID=91
   c0e4dc4b644e8bfa2a25613b60f60
11 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.

```

1 $ var1=5
2 $ env | grep var1
3 $ export var1
4 $ env | grep var1
5 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.

#### 8.4.4 `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.

```

1 $ printenv | shuf | head
2 KONSOLE_VERSION=240202
3 FZF_DEFAULT_COMMAND=fd --type f -H
4 HOME=/home/sayan
5 LANGUAGE=
6 KONSOLE_DBUS_WINDOW=/Windows/1

```

```

7 | USER=sayan
8 | CLOUDSDK_PYTHON=/usr/bin/python
9 | COLORFGBG=15;0
10 | VISUAL=nvim
11 | SSH_AUTH_SOCK=/run/user/1000/ssh-agent.socket

```

## 8.5 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.

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.

- ▶ `USER` stores the currently logged in user.<sup>2</sup>
- ▶ `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.

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.

### 8.5.1 PWD

```

1 $ echo $PWD
2 /home/sayan
3 $ cd /tmp
4 $ echo $PWD
5 /tmp

```

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

### 8.5.2 RANDOM

```

1 $ echo $RANDOM
2 11670
3 $ echo $RANDOM
4 29897

```

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

**Remark 8.5.1** The RANDOM variable is not truly random, but is a pseudo-random number gener-

ated 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](#).

### 8.5.3 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.

```
1 $ echo "echo hello" > sayhello
2 $ chmod 764 sayhello
3 mode of 'sayhello' changed from 0644 (rw-r--r
   --) to 0764 (rwxrw-r--)
4 $ sayhello
5 bash: sayhello: command not found
6 $ PATH=$PATH:$PWD
7 $ sayhello
8 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.

### 8.5.4 PS1

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

```

1 [sayan@rex ~] $ PS1="[\u@\h \w] \$ "
2 [sayan@rex ~] $ PS1="hello "
3 hello PS1="enter command> "
4 enter command> PS1="User: \u> "
5 User: sayan> PS1="User: \u, Computer: \h> "
6 User: sayan, Computer: rex> PS1="Date: \d> "
7 Date: Thu Jul 25> PS1="Time: \t> "
8 Time: 17:35:44> PS1="Jobs: \j> "
9 Jobs: 0> sleep 50 &
10 [1] 3600280
11 Jobs: 1>
12 Jobs: 1> PS1="Shell: \s> "
13 Shell: bash> PS1="History Number: \!> "
14 [1]+ Done sleep 50
15 History Number: 563>
16 History Number: 563> echo hello
17 hello
18 History Number: 564> PS1="Command Number: \#>
   "
19 Command Number: 65>
20 Command Number: 65> echo hello
21 hello
22 Command Number: 66> PS1="Ring a bell \a> "
23 Ring a bell >
24 Ring a bell > PPS1="[\u@\h \w] \$ "
25 [sayan@rex ~] $
```

These changes are temporary and are only valid for the current shell session. To make the changes per-

manent, 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.

```

1 $ prompt(){}
2 > PS1="($?) [\u@\h \w]\$ "
3 > }
4 $ PROMPT_COMMAND=prompt
5 (0)[sayan@rex ~]$ ls /home
6 sayan
7 (0)[sayan@rex ~]$ ls /random
8 ls: cannot access '/random': No such file or
   directory
9 (2)[sayan@rex ~]$
```

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

```

1 $ export PROMPT_COMMAND="
2     _RES=\${PIPESTATUS[*]};
3     _RES_STR='';
4     for res in \$_RES; do
5         if [[ ( \$res > 0 ) ]]; then
6             _RES_STR="\$_RES\$";
7         fi;
8     done"
9 $ export PS1="\u@\h \w\$_RES_STR\$ "
10 sayan@rex ~$ echo hello
11 hello
12 sayan@rex ~$ exit 1 | exit 2 | exit 3
13 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.

## 8.6 Variable Manipulation

### 8.6.1 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.

```

1 $ unset var
2 $ echo ${var:-default}
3 default
4 $ echo $var
5
6 $ var=hello
7 $ echo ${var:-default}
8 hello
9 $ echo $var
10 hello

```

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

```

1 $ unset var
2 $ echo ${var:+default}
3
4 $ var=hello
5 $ echo ${var:+default}
6 default
7 $ echo $var
8 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.<sup>3</sup> This is similar to the `:-` operator, but also sets the variable to the default value. It does nothing if the variable is already set.

<sup>3</sup>: This operator is also present in modern python and is called the walrus operator.

```

1 $ unset var
2 $ echo ${var:=default}

```

```

3 default
4 $ echo $var
5 default
6 $ var=hello
7 $ echo ${var:=default}
8 hello
9 $ echo $var
10 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 8.6.1** `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.

## 8.6.2 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.

```
1 | $ 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 privilege, 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.

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

```
1 $ unset STEAMROOT
2 $ rm -rf "${STEAMROOT:?Variable not set}"
3 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.

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.

### 8.6.3 Length of Variable

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

```
1 $ var="Hello World"
2 $ echo ${#var}
3 11
```

### 8.6.4 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.<sup>4</sup>

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.

```

1 $ var="Hello World"
2 $ echo ${var:0:5}
3 Hello
4 $ echo ${var:6:5}
5 World
6 $ echo ${var:6:50}
7 World
8 $ echo ${var: -5:5}
9 World
10 $ echo ${var: -11:5}
11 Hello

```

## 8.6.5 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"

## 8.6.6 Replace Substring

The substring of a variable can be replaced using the / operator. The syntax is \${var/pattern/string}. This will replace the first occurrence 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.

```
1 $ var="Hello World"
2 $ echo ${var/ */ Universe}
3 Hello Universe
```

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

```
1 $ var="Hello World"
2 $ echo ${var//o/O}
3 Hello WOrld
```

## 8.6.7 Anchoring Matches

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

```
1 $ var="system commands"
2 $ echo ${var/#$S/$S}
3 System commands
4 $ var="linux commands"
5 $ echo ${var/#$S/$S}
6 linux commands
```

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.

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

```
1 $ var="Hello World"
2 $ echo ${var/%?/&!}
3 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.

### 8.6.8 Deleting the match

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

```
1 | $ var="Hello World"
2 | $ 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.

### 8.6.9 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.

```
1 | $ var="sayan"
2 | $ echo ${var^}
3 | Sayan
```

Similarly, we can change the entire variable.

```
1 | $ var="SAYAN"
2 | $ echo ${var,,}
3 | sayan
```

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

```
1 | $ echo "Hello ${USER^}!"
2 | Hello Sayan!
```

### 8.6.10 Sentence Case

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

```
1 var="hELLO wORLD"
2 lower=${var,,}
3 echo ${lower^}
4 Hello world
```

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

## 8.7 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.

### 8.7.1 Integer Only

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

```
1 $ declare -i 'var'
2 $ var=5
3 $ echo "$var * $var = $((var**2))"
4 5 * 5 = 25
5 $ var=hello
6 $ echo "$var * $var = $((var**2))"
7 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.

### 8.7.2 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.

```

1 $ declare -l var
2 $ var="HELLO WORLD!"
3 $ echo $var
4 hello world!

```

### 8.7.3 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.

```

1 $ declare -u var
2 $ var="hello world!"
3 $ echo $var
4 HELLO WORLD!

```

### 8.7.4 Read Only

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

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.<sup>5</sup>

```

1 $ declare -r PI=3.14159
2 $ echo "PI = $PI"
3 PI = 3.14159
4 $ PI=3.1416
5 -bash: PI: readonly variable

```

### 8.7.5 Removing Restrictions

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

```

1 $ declare -i var
2 $ var=hello
3 $ echo $var
4 0
5 $ declare +i var
6 $ var=hello
7 $ echo $var
8 hello

```

## 8.8 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.

```

1 $ echo $-
2 himBHs

```

These can be set or unset using the set command.

```

1 $ echo $-
2 himBHs
3 $ set +m
4 $ echo $-
5 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.

```
1 | $ bash -c 'echo $-'
2 | 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 6.

## 8.9 Signals

**Remark 8.9.1** 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 be 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

## 8.10 Brace Expansion

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

**Definition 8.10.1** (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.

### 8.10.1 Range Expansion

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

**Syntax:**

```
1 | {start...end}
1 | $ echo {1..5}
2 | 1 2 3 4 5
```

We can also specify the increment value.

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

The start and end values are both inclusive.

This can also be used for alphabets.

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

### 8.10.2 List Expansion

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

**Syntax:**

```
1 | {string1,string2,string3}
```

This can be used to generate a list of strings.

```
1 | $ echo {apple,banana,cherry}
2 | apple banana cherry
```

### 8.10.3 Combining Expansions

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

```
1 $ echo file{1..5}.txt
2 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.

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

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

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

```
1 $ mkdir -p test/{a,b,c}/{1,2,3}
2 $ touch test/{a,b,c}/{1,2,3}/file{1..5}.txt
3 $ tree --charset ascii test
4 test
5   |-- a
6   |   |-- 1
7   |   |   |-- file1.txt
8   |   |   |-- file2.txt
9   |   |   |-- file3.txt
10  |   |   |-- file4.txt
11  |   |   '-- file5.txt
12  |-- 2
13  |   |-- file1.txt
14  |   |-- file2.txt
15  |   '-- file3.txt
```

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.

```
16 |     |     |-- file4.txt
17 |     |     '-- file5.txt
18 |     '-- 3
19 |         |-- file1.txt
20 |         |-- file2.txt
21 |         |-- file3.txt
22 |         |-- file4.txt
23 |         '-- file5.txt
24 |-- b
25 |     |-- 1
26 |         |-- file1.txt
27 |         |-- file2.txt
28 |         |-- file3.txt
29 |         |-- file4.txt
30 |         '-- file5.txt
31 |     |-- 2
32 |         |-- file1.txt
33 |         |-- file2.txt
34 |         |-- file3.txt
35 |         |-- file4.txt
36 |         '-- file5.txt
37 |     '-- 3
38 |         |-- file1.txt
39 |         |-- file2.txt
40 |         |-- file3.txt
41 |         |-- file4.txt
42 |         '-- file5.txt
43 '-- c
44     |-- 1
45     |     |-- file1.txt
46     |     |-- file2.txt
47     |     |-- file3.txt
48     |     |-- file4.txt
49     |     '-- file5.txt
50     |-- 2
51     |     |-- file1.txt
52     |     |-- file2.txt
53     |     |-- file3.txt
54     |     |-- file4.txt
55     |     '-- file5.txt
56     '-- 3
```

```

57      |-- file1.txt
58      |-- file2.txt
59      |-- file3.txt
60      |-- file4.txt
61      '-- file5.txt
62
63 13 directories, 45 files

```

## 8.11 History Expansion

**Definition 8.11.1** (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.

```

1 $ echo hello
2 hello
3 $ history | tail
4 499 man set
5 500 man set
6 501 man tree
7 502 tree --charset unicode
8 503 tree --charset ascii
9 504 history
10 505 tree --charset unicode
11 506 clear
12 507 echo hello
13 508 history | tail
14 $ !507
15 echo hello
16 hello

```

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

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

```
1 $ touch file1 file2 .hidden
2 $ ls
3 file1  file2
4 $ !! -a
5 ls -a
6 .  ..  .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.

```
1 $ touch /etc/test
2 touch: cannot touch '/etc/test': Permission
   denied
3 $ sudo !!
4 sudo touch /etc/test
```

## 8.12 Arrays

**Definition 8.12.1** (Array) An array is a collection of elements, each identified by an index.

We can declare an array using the declare command.

```
1 $ declare -a arr
```

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

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

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

```
1 $ arr[2]=6
2 $ echo ${arr[2]}
3 6
```

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

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

### 8.12.1 Length of Array

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

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

### 8.12.2 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.

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

### 8.12.3 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.

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

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

### 8.12.4 Deleting an Element

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

```
1 $ arr=(1 2 3 4 5)
2 $ arr[10]=6
3 $ echo ${arr[@]}
4 1 2 3 4 5 6
5 $ echo ${!arr[@]}
6 0 1 2 3 4 10
7 $ unset arr[10]
8 $ echo ${arr[@]}
9 1 2 3 4 5
10 $ echo ${!arr[@]}
11 0 1 2 3 4
```

### 8.12.5 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.

```

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

```

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

### 8.12.6 Storing output of a command in an Array

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

```

1 $ var=$(ls)
2 $ echo $var
3 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.

```

1 $ arr=($(ls))
2 $ echo ${arr[@]}
3 file1 file2 file3
4 $ echo ${!arr[@]}
5 0 1 2
6 $ echo ${arr[1]}
7 file2

```

### 8.12.7 Iterating over an Array

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

```

1 $ arr=(1 2 3 4 5)
2 $ for i in ${arr[@]}; do
3 >   echo $i
4 > done
5 1
6 2
7 3
8 4
9 5

```

We will cover loops in more detail in Chapter 9.

### 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

```

1 $ arr=("Some" "elements" "are" "multi word")
2 $ for i in "${arr[*]}"; do
3 >   echo $i
4 > done
5 Some elements are multi word

```

Here, as we are using the `*` operator, the array is expanded 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.

```

1 $ arr=("Some" "elements" "are" "multi word")
2 $ for i in ${arr[*]}; do echo $i; done
3 Some
4 elements
5 are
6 multi
7 word

1 $ arr=("Some" "elements" "are" "multi word")
2 $ for i in ${arr[@]}; do echo $i; done
3 Some
4 elements
5 are
6 multi
7 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.

```

1 $ arr=("Some" "elements" "are" "multi word")
2 $ for i in "${arr[@]}"; do echo $i; done
3 Some
4 elements
5 are
6 multi word

```

## 8.13 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.

```
1 $ declare -A arr
2 $ arr=(["name"]="Sayan" ["age"]=22)
3 $ echo ${arr[name]}
4 Sayan
5 $ arr[age]=23
6 $ echo ${#arr[@]}
7 2
8 $ echo ${!arr[@]}
9 age name
10 $ echo ${arr[@]}
11 23 Sayan
12 $ 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.

# Shell Scripting

Now that we have learnt the basics of the bash shell and the core utils, we will now see how we can combine them to create shell scripts which let us do even more complex tasks in just one command.

## 9.1 What is a shell script?

A shell script is a file that contains a sequence of shell commands. When you run a shell script, the commands in the script are executed in the order they are written. Shell scripts are used to automate tasks that would otherwise require a lot of manual work. They are also used to create complex programs that can be run from the command line.

## 9.2 Shebang

**Definition 9.2.1 (Shebang)** The shebang is a special line at the beginning of a shell script that tells the operating system which interpreter to use to run the script. The shebang is written as `#!` followed by the path to the interpreter.

For example, the shebang for a bash script is `#!/bin/bash`. However, as `/bin` is a symbolic link to `/usr/bin` in most systems, we can also specify the path as `#!/usr/bin/bash`.

The shebang is only needed if we run the script directly from the shell, without specifying the in-

terpreter. This also requires the script to have the execute permission set.

```

1 $ cat script.sh
2 #!/bin/bash
3 echo "Hello, World!"
4 $ bash script.sh
5 Hello, World!
6 $ ./script.sh
7 bash: ./script.sh: Permission denied
8 $ chmod +x script.sh
9 $ ./script.sh
10 Hello, World!
```

Here we are first running the script with the `bash` command, which is why the shebang and execute permission is not needed. However, when we try to run the script directly, we get a permission denied error as the permission is not set. We then set the execute permission and run the script again, which works.

We can also use the shebang for non-bash scripts, for example, a python script can have the shebang `#!/usr/bin/python3`.

```

1 $ cat script.py
2 #!/usr/bin/python3
3 print("Hello, World!")
4 $ python3 script.py
5 Hello, World!
6 $ chmod u+x script.py
7 $ ./script.py
8 Hello, World!
```

Even though we called the script without specifying the interpreter, the shebang tells the shell to use the `python3` interpreter to run the script. This only works if the file has the execute permission set.

If the shebang is absent, and a file is executed without specifying the interpreter, the shell will try

to execute it in its own interpreter.

## 9.3 Comments

In shell scripts, comments are lines that are not executed by the shell. Comments are used to document the script and explain what the script does. Comments in shell scripts start with a # character and continue to the end of the line.

Although it may not be required when running a command from the terminal, comments are supported there as well. Comments are most useful in scripts, where they can be used to explain what the script does and how it works.

```
1 $ echo "hello" # This is a comment  
2 $ echo "world" # we are using the echo command  
     to print the word "world"
```

Here we use the # to start a line comment. Any character after the pound sign is ignored by the interpreter.

**Remark 9.3.1** The shebang also starts with a # character, so it is also ignored by the interpreter when executing, however, if a file is executed without mentioning the interpreter then the shell reads its first line and find the path to the interpreter using the shebang

### 9.3.1 Multiline Comments

Although bash does not have a built-in syntax for multiline comments, we can use a trick to create multiline comments. We can use the NOP command

: to create a multiline comment by passing the multiline string to it.

```

1 $ : '
2 This is a multiline comment
3 This is the second line
4 This is the third line
5 '

```

## 9.4 Variables

We have already seen the various ways to create and use variables in bash. We can use these variables in scripts as well.

```

1 $ cat variables.sh
2 #!/bin/bash
3 name="alice"
4 declare -i dob=2001
5 echo "Hello, ${name}! You are $((2024 - dob))
       years old."
6 $ chmod u+x variables.sh
7 $ ./variables.sh
8 Hello, Alice! You are 23 years old.

```

The variables defined inside a script are not available outside the script if the script is executed in a new environment. However, if we want to actually retain the variables, we can use the source command to run the script in the current environment.

This reads the script and executes it in the current shell, so the variables defined in the script are available in the current shell. The PID of the script remains same as the executing shell, and no new process is created. Since this only reads the file, the execute permission is not needed.

This is useful when we want to set environment variables or aliases using a script.

```

1 $ cat variables.sh
2 name="alice"
3 dob=2001
4 $ source variables.sh
5 $ echo "Hello, ${name}! You are $((2024 - dob
   )) years old."
6 Hello, Alice! You are 23 years old.

```

Similarly, if we want to run a script in a new environment, but we still want it to have read access to the variables declared in the current environment, we can export the variables needed.

```

1 $ cat variables.sh
2 #!/bin/bash
3 echo "Hello, ${name}! You are $((2024 - dob))
   years old."
4 $ chmod u+x variables.sh
5 $ ./variables.sh
6 Hello, ! You are 2024 years old.
7 $ name="alice"
8 $ dob=2001
9 $ ./variables.sh
10 Hello, ! You are 2024 years old.
11 $ export name dob
12 $ ./variables.sh
13 Hello, Alice! You are 23 years old.

```

In this example we can see that if a variable is not defined in the script and we access the variable in the script, it will take an empty string. If we define the variable in the parent shell and then call the script, it will still not have access to it. However, if we export the variable, then calling the script will provide it with the variable.

This is because `export` makes the variable into an environment variable, which is available to all child processes of the current shell. When a new environment is created, the environment variables are copied to the new environment, so the script has access to the exported variables.

However, if we want to add some variable to the environment of the script, but we do not want to add it to our current shell's environment, we can directly specify the variable and the value before running the script, in the same line. This sets the

variables for the script's environment, but not for the shell's environment.

```

1 $ cat variables.sh
2 #!/bin/bash
3 echo "Hello, ${name}! You are $((2024 - dob))
   years old."
4 $ ./variables.sh
5 Hello, ! You are 2024 years old.
6 $ name=Alice dob=2001 ./script.sh
7 Hello, Alice! You are 23 years old.

```

## 9.5 Arguments

Just like we can provide arguments to a command, we can also provide arguments to a script. These arguments are stored in special variables that can be accessed inside the script.

- ▶ `$0` - The path of the script/interpreter by which it is called.
- ▶ `$1` - The first argument
- ▶ `$2` - The second argument
- ⋮
- ▶ `$@` - All arguments stored as an array
- ▶ `$*` - All arguments stored as a space separated string
- ▶ `$#` - The number of arguments

```

1 $ cat arguments.sh
2 echo $0
3 $ ./arguments.sh
4 ./arguments.sh
5 $ bash arguments.sh
6 arguments.sh
7 $ source arguments.sh
8 /bin/bash
9 $ echo $0

```

```
10 | /bin/bash
```

If we know the number of arguments that will be passed to the script, we can directly access them using the \$n syntax.

```
1 | $ cat arguments.sh
2 | echo $1 $2
3 | echo $3
4 | $ bash arguments.sh Hello World "This is a
   multiword single argument"
5 | Hello World
6 | This is a multiword single argument
```

Here we can also see how to pass multiple words as a single argument, by quoting it.

Sometimes, we may want to pass the value of a variable as an argument, in those cases, we should always quote the variable expansion, as if the variable contains multiple words, bash will expand the variables and treat each word as a separate argument, instead of the entire value of the variable as a single argument.

Quoting the variable prevents word splitting.

```
1 | $ cat split.sh
2 | echo $1
3 | $ var="Hello World"
4 | $ bash split.sh $var
5 | Hello
6 | $ bash split.sh "$var"
7 | Hello World
```

The first attempt prints only `Hello` as the second word is stored in `$2`. But if we quote the variable expansion, then the entire string is taken as one parameter.

## Double vs Single Quotes

There are some subtle differences between double and single quotes. Although both are used to prevent word splitting, single quotes also prevent variable expansion, while double quotes do not.

```

1 $ echo "Hello $USER"
2 Hello sayan
3 $ echo 'Hello $USER'
4 Hello $USER

```

However, if we do not know the number of arguments that will be passed to the script, we can use the \$@ and \$\* variables to access all the arguments.

```

1 $ cat all-arguments.sh
2 echo "$@"
3 $ bash all-arguments.sh Hello World "This is a
   multiword single argument"
4 Hello World This is a multiword single
   argument

```

Here we can see that \$@ expands to all the arguments as an array, while \$\* expands to all the arguments as a single string. In this case, the output of both looks same as echo its arguments side by side, separated by a space.

We can observe the difference between \$@ and \$\* when we use them in a loop, or pass to any command that shows output for each argument on separate lines.

```

1 $ cat multiword.sh
2 touch "$@"
3 $ bash multiword.sh Hello World "This is a
   multiword single argument"
4 $ ls -1
5 Hello
6 multiword.sh
7 'This is a multiword single argument'

```

8 | World

In this case, since we are using "\$@" (the quoting is important), the touch command is called with each argument as a separate argument, so it creates a file for each argument. The last array element has multiple words, but it creates a single file with the entire sentence as the file name.

Whereas if we use \$\*, the arguments are stored as a string, and it is split by spaces, so the touch command will create a file for each word.

```

1 $ cat multiword.sh
2 touch $*
3 $ bash multiword.sh Hello World "This is a
   multiword single argument"
4 $ ls -1
5 a
6 argument
7 Hello
8 is
9 multiword
10 multiword.sh
11 single
12 This
13 World

```

In this case, the touch command is called with each word as a separate argument, so it creates a file for each word.

**Exercise 9.5.1** Similarly, try out the other way of iterating over the arguments ("\$\*") and observe the difference.

We can also iterate over the indices and access each argument using a for loop. To find the number of arguments, we can use \$#.

```

1 $ cat args.sh

```

```

2 args=("$@")
3 echo $#
4 for ((i=0; i < $#; i++)); do
5   echo "${args[i]}"
6 done
7 $ bash args.sh hello how are you
8 4
9 hello
10 how
11 are
12 you

```

Here we are using `$#` to get the number of arguments the script has received and then dynamically iterating that many times to access each element of the `args` array.

### 9.5.1 Shifting Arguments

Sometimes we may want to remove the first few arguments from the list of arguments. We can do this using the `shift` command.

`shift` is a shell builtin that shifts the arguments to the left by one. The first argument is removed, and the second argument becomes the first argument, the third argument becomes the second argument, and so on.

```

1 $ cat shift.sh
2 echo "First argument is $1"
3 shift
4 echo "Rest of the arguments are $*"
5 $ bash shift.sh one two three four
6 First argument is one
7 Rest of the arguments are two three four

```

## 9.6 Input and Output

Now that we have seen how to pass arguments to a script, we will see how to take input from the user and display output to the user.

This is called the standard streams in bash. There are three standard streams in bash:

- ▶ **Standard Input (stdin)** - This is the input stream that is used to read input from the user. By default, this is the keyboard.
- ▶ **Standard Output (stdout)** - This is the output stream that is used to display output to the user. By default, this is the terminal.
- ▶ **Standard Error (stderr)** - This is the error stream that is used to display error messages to the user. By default, this is the terminal.

The streams can also be redirected to/from files or other commands, as we have seen in Chapter 5.

### 9.6.1 Reading Input

To read input from the user, we can use the `read` command. The `read` command reads a line of input from the `stdin` and stores it in a variable.

Optionally, we can also provide a prompt to the user, which is displayed before the user enters the input.

```

1 $ read -p "Enter your name: " name
2 Enter your name: Alice
3 $ echo "Hello, $name!"
4 Hello, Alice!

```

This reads the input from the user and stores it in the `name` variable, which is then used to display a greeting message.

We can use the same command inside a script to read input from the user, or take in any data passed in standard input.

```

1 $ cat input.sh
2 read line
3 echo "STDIN: $line"
4 $ echo "Hello, World!" | bash input.sh
5 STDIN: Hello, World!

```

However, the `read` command reads only one line of input. If we want to read multiple lines of input, we can use a loop to read input until the user enters a specific keyword.

```

1 $ cat multiline.sh
2 while read line; do
3   echo "STDIN: $line"
4 done
5 $ cat file.txt
6 Hello World
7 This is a multiline file
8 We have three lines in it
9 $ cat file.txt | bash multiline.sh
10 STDIN: Hello World
11 STDIN: This is a multiline file
12 STDIN: We have three lines in it

```

Here we are reading input from the file `file.txt` and displaying each line of the file.

The redirection can be done more succinctly by using the input redirection operator `<`. The pipe was used to demonstrate that the input can be coming from the output of any arbitrary commands, not just the content of a file.

```

1 $ bash multiline.sh < file.txt
2 STDIN: Hello World
3 STDIN: This is a multiline file
4 STDIN: We have three lines in it

```

## 9.7 Conditionals

Often times, in scripts, we require to decide between two blocks of code to execute, depending on some non-static condition. This can be either based on some value inputted by the user, a value of a file on the filesystem, or some other value fetched from the sensors or over the internet. To facilitate branching in the scripts, bash provides multiple keywords and commands.

### 9.7.1 Test command

The `test` command is a command that checks the value of an expression and either exits with a exit code of 0<sup>1</sup> if the expression is true, or exits with a non-zero exit code if the expression is false.

It has a lot of unary and binary operators that can be used to check various conditions.

```
1 $ test 1 -eq 1
2 $ echo $?
3 0
4 $ test 5 -gt 7
5 $ echo $?
6 1
```

<sup>1</sup>: Exit code 0 denotes success in POSIX

Here we evaluate two equations,  $1 = 1$  and  $5 > 7$ . The first equation is true, so the `test` command exits with a 0 exit code, while the second equation is false, so the `test` command exits with a 1 exit code.

### String conditions

`test` can check for unary and binary conditions on strings.

- ▶ `-z` - True if the string is empty
- ▶ `-n` - True if the string is not empty
- ▶ `=` - True if the strings are equal
- ▶ `!=` - True if the strings are not equal

- ▶ < - True if the first string is lexicographically less than the second string
- ▶ > - True if the first string is lexicographically greater than the second string

## Unary Operators

The `-z` and `-n` flags of `test` check if a string is empty or not.

```

1 $ var="apple"
2 $ test -n "$var" ; echo $?
3 0
4 $ test -z "$var" ; echo $?
5 1

1 $ var=""
2 $ test -n "$var" ; echo $?
3 1
4 $ test -z "$var" ; echo $?
5 0

```

## Binary Operators

The `=`, `!=`, `<`, and `>` flags of `test` check if two strings are equal, not equal, less than, or greater than each other.

```

1 $ var="apple"
2 $ test "$var" = "apple" ; echo $?
3 0
4 $ test "$var" != "apple" ; echo $?
5 1

```

```

1 $ var="apple"
2 $ test "$var" = "banana" ; echo $?
3 1
4 $ test "$var" != "banana" ; echo $?
5 0

```

We are escaping the `>` and `<` characters as they have special meaning in the shell. If we do not escape them, the shell will try to redirect the input or output of the command to/from a file. We can also quote the symbols instead of escaping them.

```

1 $ var="apple"
2 $ test "$var" \< "banana" ; echo $?
3 0

```

```

4 $ test "$var" \> "banana" ; echo $?
5 1

```

## Numeric conditions

`test` can also check for unary and binary conditions on numbers.

- ▶ `-eq` - True if the numbers are equal
- ▶ `-ne` - True if the numbers are not equal
- ▶ `-lt` - True if the first number is less than the second number
- ▶ `-le` - True if the first number is less than or equal to the second number
- ▶ `-gt` - True if the first number is greater than the second number
- ▶ `-ge` - True if the first number is greater than or equal to the second number

```

1 $ test 5 -eq 5 ; echo $?
2 0
3 $ test 5 -ne 5 ; echo $?
4 1
5 $ test 5 -ge 5 ; echo $?
6 0
7 $ test 5 -lt 7 ; echo $?
8 0
9 $ test 5 -le 7 ; echo $?
10 0
11 $ test 5 -gt 7 ; echo $?
12 1
13 $ test 5 -ge 7 ; echo $?
14 1

```

## File conditions

The most number of checks in `test` are for files. We can check if a file exists, if it is a directory, if it is a

regular file, if it is readable, writable, or executable, and many more.

### File Types

- ▶ -e and -a - if the file exists
- ▶ -f - if the file is a regular file
- ▶ -d - if the file is a directory
- ▶ -b - if the file is a block device
- ▶ -c - if the file is a character device
- ▶ -h and -L - if the file is a symbolic link
- ▶ -p - if the file is a named pipe
- ▶ -s - if the file is a socket

### File Permissions

- ▶ -r - if the file is readable
- ▶ -w - if the file is writable
- ▶ -x - if the file is executable
- ▶ -u - if the file has the setuid bit set
- ▶ -g - if the file has the setgid bit set
- ▶ -k - if the file has the sticky bit set
- ▶ -o - if the file is effectively owned by the user
- ▶ -G - if the file is effectively owned by the user's group

### Binary Operators

- ▶ -nt - if the first file is newer than the second file
- ▶ -ot - if the first file is older than the second file
- ▶ -ef - if the first file is a hard link to the second file

### Other conditions

- ▶ -o - Logical OR (When used in binary operation)
- ▶ -a - Logical AND

- ▶ ! - Logical NOT
- ▶ -v - if the variable is set
- ▶ -o - if the shopt option is set (when used as unary operator)

The test built-in can be invoked in two ways, either by using the test word or by using the [ built-in. The [ is a synonym for the test command, and is used to make the code more readable. If we use the [ command, the last argument must be ]. This makes it look like a shell syntax rather than a command, but we should keep in mind that the test command (even the [ shortcut) is a command built-in, and not a shell keyword, thus we need to use a space after the command.

```

1 $ [ 1 -eq 1 ] ; echo $?
2 0
3 $ var=apple
4 $ [ "$var" = "apple" ] ; echo $?
5 0

```

**Remark 9.7.1** The test and [ exist as both a executable stored in the filesystem and also as a shell built-in in bash. However the built-in takes preference when we simply type the name. Almost all operators are same in both, however the executable version cannot check if a variable is set using the -v option since executables dont have access to the shell variables.

## 9.7.2 Test Keyword

Although we already have the test executable and its shorthand [ as built-in, bash also provides a keyword [[ which is a more powerful version of the test command. This is present in bash<sup>2</sup> but not

2: And other popular shells like zsh

present in the POSIX standard.

```

1 $ type -a test
2 test is a shell builtin
3 test is /usr/bin/test
4 $ type -a [
5 [ is a shell builtin
6 [ is /usr/bin/[
7 $ type -a [[
8 [[ is a shell keyword

```

This is an improvement over the test command, as it has more features and is more reliable.

- ▶ It does not require us to quote variables
- ▶ It supports logical operators like && and ||
- ▶ It supports regular expressions
- ▶ It supports globbing

## Quoting and Empty Variables

```

1 $ var=""
2 $ [ $var = "apple" ] ; echo $?
3 -bash: [: =: unary operator expected
4 2
5 $ [ "$var" = "apple" ] ; echo $?
6 1

```

As we can see, the test command requires us to quote the variables, otherwise it will throw an error if the variable is empty. However, the [[ keyword does not require us to quote the variables.

```

1 $ var=""
2 $ [[ $var = "apple" ]] ; echo $?
3 1

```

## Logical Operators

The logical operators to combine multiple conditions are && and ||. In the test command, we can use the -a and -o operators, or we can use && and || outside by combining multiple test commands.

```

1 $ var="apple"
2 $ [ "$var" = "apple" -a 1 -eq 1 ] ; echo $?
3 0
4 $ [ "$var" = "apple" ] && [ 1 -eq 1 ] ; echo $?
5 0

```

But in the [[ keyword, we can use the && and || operators directly.

```

1 $ var="apple"
2 $ [[ "$var" = "apple" && 1 -eq 1 ]] ; echo $?
3 0

```

## Comparison Operators

For string comparison, we had to escape the > and < characters in the test command, but in the [[ keyword, we can use them directly.

```

1 $ var="apple"
2 $ [[ "$var" < "banana" ]] ; echo $?
3 0

```

## Regular Expressions

The [[ keyword also supports regular expressions using the =~ operator.

If we want to check whether the variable response is y or yes, in test we would do the following.

```

1 $ response="yes"
2 $ [[ "$response" = "y" -o "$response" = "yes" ]]
   ; echo $?
3 0

```

But in [[ we can use the =~ operator to check if the variable matches a regular expression.

```

1 $ response="yes"
2 $ [[ "$response" =~ ^y(es)?$ ]] ; echo $?
3 0

```

It also uses **ERE** (Extended Regular Expressions) by default, so we can use the ? operator without escaping it to match 0 or 1 occurrence of the previous character.

**Warning 9.7.1** When using regular expressions in bash scripts, we should never quote the regular expression, as it will be treated as a string and not a regular expression.

## Globbing

We can also simply match for any response starting with y using globbing.

```
1 $ response="yes"
2 $ [[ "$response" == y* ]] ; echo $?
3 0
```

## Double Equals

In the [[ keyword, we can use the == operator to check for string equality, which is a widely known construct from most languages.

## 9.8 If-elif-else

### 9.8.1 If

The if statement is used to execute a block of code if a condition is true. If the condition is false, the block of code is skipped.

The end of the if statement is denoted by the fi keyword, which is the reverse of the if keyword.

The syntax of the if statement is as follows:

```
1 if command ; then
2   # code to execute if the command is
      successful
3 else
```

```

4 # code to execute if the command is not
   successful
5 fi

```

Unlike most programming language that takes a boolean expression, the `if` statement in bash takes a command, and executes the command. If the command exits with a 0 exit code, the block of code after the `then` keyword is executed, otherwise the block of code after the `else` keyword is executed.

This allows us to use any command that we have seen so far in the `if` statement. The most used command is the `test` command, which is used to check conditions.

```

1 $ var="apple"
2 $ if [ "$var" =~ ^a.* ] ; then
3 >   echo "The variable starts with a"
4 > fi
5 The variable starts with a

```

However, we can also use other commands, like the `grep` command, to check if a file contains a pattern.

```

1 $ cat file.txt
2 Hello how are you
3 $ if grep -q "how" file.txt ; then
4 >   echo "The file contains the word how"
5 > fi
6 The file contains the word how
7 $ if grep -q "world" file.txt ; then
8 >   echo "The file contains the word world"
9 > fi

```

The first `if` statement checks if the file contains the word `how`, which is present, and thus prints the message to the screen. The second `if` statement checks if the file contains the word `world`, which is not present, and thus does not print anything.

### 9.8.2 Else

The `else` keyword is used to execute a block of code if the command in the `if` statement is not successful.

```

1 $ var="apple"
2 $ if [ "$var" = "banana" ] ; then
3 >   echo "The variable is banana"
4 > else
5 >   echo "The variable is not banana"
6 > fi
7 The variable is not banana

```

The combination of the `if` and `else` keywords allows us to execute different blocks of code depending on the condition; this is one of the most used construct in scripting.

```

1 $ read -p "Enter a number: " num
2 Enter a number: 5
3 $ if [[ $num -gt 0 ]] ; then
4 >   echo "The number is positive"
5 > else
6 >   echo "The number is negative"
7 > fi
8 The number is positive

```

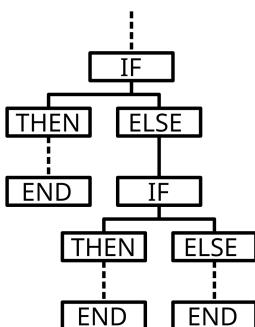


Figure 9.1: Flowchart of the `if`, `elif`, and `else` construct

However, notice that this allows us only two branches, one for the condition being true, and one for the condition being false. Due to this, we have misclassified the number 0 as positive, which is not correct.

To fix this, we can use the `elif` keyword, which is short for `else if`.

### 9.8.3 Elif

If the command in the `if` statement is not successful, we can run another command using the `elif` keyword and decide the branching based on its exit status.

```

1 $ read -p "Enter a number: " num
2 Enter a number: 0
3 $ if [[ $num -gt 0 ]] ; then
4 >   echo "The number is positive"
5 > elif [[ $num -lt 0 ]] ; then
6 >   echo "The number is negative"
7 > else
8 >   echo "The number is zero"
9 > fi
10 The number is zero

```

Here we have added an `elif` statement to check if the number is less than 0, and if it is, we print that the number is negative, finally if both the commands are unsuccessful, we execute the statements in the `else` block.

## 9.9 Exit code inversion

We can also invert the condition using the `!` operator.

```

1 $ var=apple
2 $ if ! [ "$var" = "banana" ] ; then
3 >   echo "The variable is not banana"
4 > fi
5 The variable is not banana

```

## 9.10 Mathematical Expressions as if command

We can also use the `((` keyword to evaluate mathematical expressions in the `if` statement. This environment does not print any output to the standard output, rather, it exits with a zero exit code if the result of the mathematical evaluation is non-zero, and exits with a non-zero exit code if the evaluation results to zero. This is useful as 0 is false in mathematics but 0 is success exit code in the shell.

```

1 $ if (( 5 > 3 )) ; then
2 >   echo "5 is greater than 3"
3 > fi
4 5 is greater than 3

```

However, this environment only supports integers, and not floating point numbers.

## 9.11 Command Substitution in if

We can also run a command in the `if` statement, and compare the output of the command to a value using the `test` command and the `$()` construct.

```

1 $ if [ $(wc -l < file.txt) -gt 100 ] ; then
2 >   echo "The file is big"
3 > fi
4 The file is big

```

## 9.12 Switch

**Definition 9.12.1** (Switch Case) A switch case is a syntactical sugar that helps match a variable's

value against a list of options (or patterns) and executes the block of code corresponding to the match.

The branching achieved by **switch case** can also be achieved by using multiple **if-elif-else** statements, but the switch case is more readable and concise.

## Syntax

```

1 case var in
2   pattern1)
3     # code to execute if var matches pattern1
4     ;;
5   pattern2)
6     # code to execute if var matches pattern2
7     ;;
8   *)
9     # code to execute if var does not match
10    any pattern
11    ;;
11 esac
```

The **case** block is ended with the keyword **esac**, which is the reverse of the **case** keyword.

The **case** keyword is followed by the variable to match against, and the **in** keyword. Then we have multiple patterns, each followed by a block of code to execute if the variable matches the pattern.

The **;;** keyword is used to denote the end of a block of code, and the start of the next pattern. This ensures that the script does not continue running the statements of the other blocks after it as well.

The **\*** pattern is a wildcard pattern, and matches any value that does not match any of the other patterns.

The patterns in the switch case are similar to globs rather than regular expressions, so we cannot use regular expressions in the switch case.

Although sometimes we actually want to execute the blocks after the match. This is called fall through.

### 9.12.1 Fall Through

In this example, the script will print both the statements "Multiple of 10" and "Multiple of 5" for numbers ending with zero because of the fall through ;& instead of ;;.

```

1 $ cat switch.sh
2 read -p "Enter number: " num
3 case "$num" in
4 *) echo "Multiple of 10" ;&
5 *) echo "Multiple of 5" ;;
6 *) echo "Not Multiple of 5" ;;
7 esac
8 $ bash switch.sh
9 Enter number: 60
10 Multiple of 10
11 Multiple of 5

```

### 9.12.2 Multiple Patterns

We can also have multiple patterns for a single block of code by separating the patterns with the | operator.

```

1 $ cat switch.sh
2 read -p "Enter digit: " num
3 case "$num" in
4|2|3) echo "Small number" ;;
5|5|6) echo "Medium number" ;;
6|7|8|9) echo "Large number" ;;
7*) echo "Invalid number" ;;
8 esac
9 $ bash switch.sh
10 Enter digit: 5
11 Medium number

```

## 9.13 Select Loop

**Definition 9.13.1** (Select Loop) A select loop is a construct that is used to create a menu in the shell script. It takes a list of items and displays them to the user, and then waits for the user to select an item. The selected item is stored in a variable, and the block of code corresponding to the selected item is executed. This repeats infinitely, until the user stops it.

```

1 $ cat select.sh
2 select choice in dog cat bird stop
3 do
4     case $choice in
5         dog) echo "A dog barks" ;;
6         cat) echo "A cat meows" ;;
7         bird) echo "A bird chirps" ;;
8         stop) break ;;
9         *) echo "Invalid choice" ;;
10    esac
11 done
12 $ bash select.sh
13 1) dog
14 2) cat
15 3) bird
16 4) stop
17 #? 1
18 A dog barks
19 #? 2
20 A cat meows
21 #? 3
22 A bird chirps
23 #? 4

```

A select loop is simply a syntactical sugar for a while loop that displays a menu to the user and waits for the user to select an option and reads it using `read`. The conditional branching is done using

a case statement.

The menu output is actually displayed on the standard error stream, so that it is easy to split the conditional output from the menu output.

```
1 $ bash select.sh > output.txt
2 1) dog
3 2) cat
4 3) bird
5 4) stop
6 #? 1
7 #? 2
8 #? 3
9 #? 4
10 $ cat output.txt
11 A dog barks
12 A cat meows
13 A bird chirps
```

To stop the select loop, we can use the `break` keyword. We will see more about loops in the next section.

## 9.14 Loops

If we want to execute a block of code multiple times, we can use loops. There are three types of loops in bash:

- ▶ **For loop** - Used to iterate over a list of items or a fixed number of times
- ▶ **While loop** - Used to execute a block of code as long as a condition is true
- ▶ **Until loop** - Used to execute a block of code as long as a condition is false

## 9.14.1 For loop

In bash, we can use the `for` loop to iterate over a list of items, called a **for-each** loop<sup>3</sup>, or a range of numbers<sup>4</sup>.

### For-each loop

```

1 $ cat foreach.sh
2 for item in mango banana strawberry; do
3   echo "$item shake"
4 done
5 $ bash foreach.sh
6 mango shake
7 banana shake
8 strawberry shake

```

3: This is similar to how the `for` loop in Python works.

4: This is similar to how the `for` loop in C and C like languages works.

As we saw in Chapter 8, there are three ways to iterate over an array in bash. We can either treat the entire array as a single element (" `${arr[*]}` "), we can break the elements by spaces ( `${arr[@]}` ), or we can split as per the array elements, preserving multi-word elements (" `${arr[@]}` ").

#### Treating entire array as a single element:

```

1 $ cat forsplit.sh
2 name=("Sayan" "Alice" "John Doe")
3 for name in "${name[*]}"; do
4   echo "Name: $name"
5 done
6 $ bash forsplit.sh
7 Name: Sayan Alice John Doe

```

#### Splitting by spaces:

```

1 $ cat forsplit.sh
2 name=("Sayan" "Alice" "John Doe")
3 for name in ${name[@]}; do
4   echo "Name: $name"
5 done

```

```

6 $ bash forsplit.sh
7 Name: Sayan
8 Name: Alice
9 Name: John
10 Name: Doe

```

### Preserving multi-word elements:

```

1 $ cat forsplit.sh
2 name=("Sayan" "Alice" "John Doe")
3 for name in "${name[@]}"; do
4   echo "Name: $name"
5 done
6 $ bash forsplit.sh
7 Name: Sayan
8 Name: Alice
9 Name: John Doe

```

We can also dynamically generate a list of numbers in a range using the {start..end} syntax or the seq command.

```

1 $ cat range.sh
2 for i in {1..5}; do
3   echo "Number: $i"
4 done
5 $ bash range.sh
6 1
7 2
8 3
9 4
10 5

1 $ cat range.sh
2 for i in $(seq 1 5); do
3   echo "Number: $i"
4 done
5 $ bash range.sh
6 1
7 2
8 3
9 4

```

10 | 5

## Differences between range expansion and seq command

Although both seq and the **range expansion** bash syntax have similar functionality, they have slightly different behaviour, as seen in Table 9.1.

**Table 9.1:** Differences between range expansion and seq command

Range Expansion	Seq Command
It is a bash feature hence it is faster	It is an external command
Only integer step size	Fractional step size is allowed
Works on letters	Works only on numbers
Step size should be the third argument	Step size should be the second argument
Output is space separated	Output is newline separated
Start range cannot be omitted	Start range is 1 by default

### Letters:

```

1 $ echo {a..e}
2 a b c d e
3 $ seq a e
4 seq: invalid floating point argument: 'a'
5 Try 'seq --help' for more information.

```

### Position of step size:

```

1 $ echo {1..10..2}
2 1 3 5 7 9
3 $ seq 1 2 10
4 1
5 3
6 5
7 7
8 9

```

### Fractional step size:

```

1 $ seq 1 0.5 2
2 1.0
3 1.5

```

Note that if the shell is not able to expand a range expansion due to invalid syntax it will not throw an error, rather it will simply keep it unexpanded. This is similar to how path expansion works.

```

4| 2.0
5| $ echo {1..2..0.5}
6| {1..2..0.5}

```

### Default start range:

```

1| $ seq 5
2| 1
3| 2
4| 3
5| 4
6| 5

```

### Different between python's range and seq command:

The seq command is similar to the range function in Python, but there are some differences between the two.

**Table 9.2:** Differences between seq and python's range function

Python's Range	Seq Command
Start of range is 0 by default	Start range is 1 by default
Order of parameters is start,end,step	Order of parameters is start,step,end
End range is exclusive	End range is inclusive

## 9.14.2 C style for loop

Bash also supports C-style for-loops, where we declare a variable, a condition for the loop, and an increment command.

```

1| $ cat cstyle.sh
2| for ((i=0; i<5; i++)); do
3|   echo "Number: $i"
4| done
5| $ bash cstyle.sh

```

```

6 Number: 0
7 Number: 1
8 Number: 2
9 Number: 3
10 Number: 4

```

We can also have multiple variables in the C-style for loop.

```

1 $ cat cstyle.sh
2 begin1=1
3 begin2=10
4 finish=10
5 for (( a=$begin1, b=$begin2; a < $finish; a++,
       b-- )); do
6   echo $a $b
7 done
8 $ bash cstyle.sh
9 1 10
10 2 9
11 3 8
12 4 7
13 5 6
14 6 5
15 7 4
16 8 3
17 9 2

```

### 9.14.3 IFS

By default the for loop splits the input by tabs, spaces, and newlines. We can change the delimiter by changing the `IFS` variable.

**Definition 9.14.1 (IFS)** The `IFS` variable is the Internal Field Separator, and is used to split the input into fields.

5: The `$''` syntax is used to denote ANSI escape sequences in the string in bash.

In this example we are reading the `$PATH` variable, which is a colon separated list of directories. We are changing the `IFS` variable to a colon, so that the for loop splits the input by colon.

It is used by for loop and other word splitting operations in bash.

Default value of `IFS` is `' \t\n'`. If set to multiple characters, it will split the input by any of the characters.<sup>5</sup>

```
1 $ cat ifs.sh
2 IFS=:
3 for i in $PATH; do
4     echo $i
5 done
6 $ bash ifs.sh
7 /usr/local/sbin
8 /usr/local/bin
9 /usr/bin
```

We should remember to reset the `IFS` variable after using it, as it can cause unexpected behaviour in other parts of the script.

```
1 $ cat unsetifs.sh
2 IFS=:
3 # some code
4
5 var="a b c:d"
6 for i in $var; do
7     echo $i
8 done
9 $ bash unsetifs.sh
10 a b c
11 d
```

Although we wanted to iterate over the elements by splitting by space, we ended up splitting by colon because we forgot to reset the `IFS` variable.

To reset the `IFS` variable, we can simply unset it.

```
1 $ cat unsetifs.sh
2 IFS=:
3 # some code
4
```

```

5 unset IFS
6 var="a b c:d"
7 for i in $var; do
8   echo $i
9 done
10 $ bash unsetifs.sh
11 a
12 b
13 c:d

```

Unsetting the `IFS` variable will reset it to the default value of `' \t\n'`.

However, setting and resetting the `IFS` variable can be cumbersome, so we can use a subshell to change the `IFS` variable only for the `for` loop.

```

1 $ cat subshellifs.sh
2 var="a b c:d"
3 (
4 IFS=:
5 for i in $var; do
6   echo $i
7 done
8 )
9
10 for i in $var; do
11   echo $i
12 done
13 $ bash subshellifs.sh
14 a b c
15 d
16 a
17 b
18 c:d

```

#### 9.14.4 While loop

The `while` loop is used to execute a block of code as long as a condition is true. This is useful if we do not

know the number of iterations beforehand, rather we know the condition that should be satisfied.

Just like the `if` statement, the `while` loop also takes a command, and executes the block of code if the command is successful. This means we can run any arbitrary command in the `while` loop condition checking.

```

1 $ cat while.sh
2 i=5
3 while [ $i -gt 0 ]; do
4   echo "i is $i"
5   ((i--))
6 done
7 $ bash while.sh
8 i is 5
9 i is 4
10 i is 3
11 i is 2
12 i is 1

```

Here we are using the `test` command to check if the variable `i` is greater than 0, and if it is, we print the value of `i` and decrement it by 1.

However, `test` is not the only command that we can use in the `while` loop condition.

Here we are using the `grep` command to check if the password contains any special character or not, and looping as long as the user enters a strong password.

Ideally when reading passwords, we should use the `read -s` command to hide the input from the user, however it is omitted from the example so that the input can be seen by the reader.

```

1 $ cat pass.sh
2 read -p "Enter password: " pass
3
4 while ! grep -q '[[[:punct:]]]' <<< "$pass" ; do
5   echo "Password must contain at least one
6       special character"
7   read -p "Enter password: " pass
8 done
9 echo "Password is set"
$ bash pass.sh
Enter password: 123
Password must contain at least one special
character

```

```
12 Enter password: abc
13 Password must contain at least one special
   character
14 Enter password: sayan
15 Password must contain at least one special
   character
16 Enter password: $ayan
17 Password is set
```

### 9.14.5 Until loop

The until loop is used to execute a block of code as long as a condition is false. This is simply a negation of the while loop. It is a syntactical sugar. The last example we saw in the while loop can be rewritten using the until loop to omit the ! symbol.

```
1 $ cat pass.sh
2 read -p "Enter password: " pass
3
4 until grep -q '[[[:punct:]]]' <<< "$pass" ; do
5   echo "Password must contain at least one
     special character"
6   read -p "Enter password: " pass
7 done
8 echo "Password is set"
9 $ bash pass.sh
10 Enter password: 123
11 Password must contain at least one special
   character
12 Enter password: abc
13 Password must contain at least one special
   character
14 Enter password: sayan
15 Password must contain at least one special
   character
16 Enter password: $ayan
17 Password is set
```

### 9.14.6 Read in while

6: The end of input is denoted by the EOF character, if we are reading input from standard input, we can press **Ctrl+D** to send the EOF character and mark the end of input.

The `read` command is used to read one line of input from the user, however if we want to read multiple lines of input, and we do not know the number of lines beforehand, we can use the `while` loop to read input until the user enters a specific value, or the input stops.<sup>6</sup>

```

1 $ cat read.sh
2 while read line; do
3     echo "Line is $line"
4 done
5 $ bash read.sh
6 hello
7 Line is hello
8 this is typed input
9 Line is this is typed input
10 now press ctrl+D
11 Line is now press ctrl+D

```

Here we are reading the `/etc/passwd` file line by line and printing the username. The username is the first field in the `/etc/passwd` file, which is separated by a colon. The username is extracted by removing everything after the first colon using shell variable manipulation.

We can also read from a file using the `<` operator. Since the entire while loop is a command, we can use the `<` operator to redirect the input to the while loop after the ending `done` keyword.

```

1 $ cat read.sh
2 while read line; do
3     echo ${line%%:*}
4 done < /etc/passwd
5 $ bash read.sh
6 root
7 bin
8 daemon
9 mail
10 ftp
11 http
12 nobody
13 dbus
14 systemd-coredump
15 systemd-network

```

We can also use the `IFS` variable to split each row by a colon, and extract the username.

```

1 $ cat read.sh
2 while IFS=: read username pass uid gid gecos
3   home shell; do
4   echo $username - ${gecos:-$username}
5 done < /etc/passwd
6 $ bash read.sh
7 root - root
8 bin - bin
9 daemon - daemon
10 mail - mail
11 ftp - ftp
12 http - http
13 nobody - Kernel Overflow User
14 dbus - System Message Bus
15 systemd-coredump - systemd Core Dumper
16 systemd-network - systemd Network Management

```

If the number of variables provided to the `read` command is less than the number of fields in the input, the remaining fields are stored in the last variable. This can be utilized to read only the first field of the input, and discard the rest.

```

1 $ cat read.sh
2 while IFS=: read username _; do
3   echo $username
4 done < /etc/passwd
5 $ bash read.sh
6 root
7 bin
8 daemon
9 mail
10 ftp
11 http
12 nobody
13 dbus
14 systemd-coredump
15 systemd-network

```

In this example, we are setting the `IFS` only for the `while` loop, and it gets reset after the loop. If we provide multiple variables to the `read` command, it will split the input by the `IFS` variable and assign the split values to the variables.

In this example, we are reading only the first field of the input, and discarding the rest. In bash (and many other languages) the underscore variable is used to denote a variable that is not to be used.

### 9.14.7 Break and Continue

The `break` and `continue` keywords are used to control the flow of the loop.

- ▶ **Break** - The `break` keyword is used to exit the loop immediately. This skips the current iteration, and also all the next iterations.
- ▶ **Continue** - The `continue` keyword is used to skip the rest of the code in the loop and go to the next iteration.

#### Break:

```

1 $ cat pat.sh
2 for i in {1..5}; do
3   for j in {1..5}; do
4     if [[ "$j" -eq 3 ]]; then
5       break;
6     fi
7     echo -n $j
8   done
9   echo
10 done
11 $ bash pat.sh
12 12
13 12
14 12
15 12
16 12

```

#### Continue:

```

1 $ cat pat.sh
2 for i in {1..5}; do
3   for j in {1..5}; do
4     if [[ "$j" -eq 3 ]]; then
5       continue;

```

```

6      fi
7      echo -n $j
8  done
9  echo
10 done
$ bash pat.sh
1245
1245
1245
1245
1245
1245

```

Unlike most languages, the `break` and `continue` keywords in bash also takes an optional argument. This argument is the number of loops to break or continue. If we change the `break` keyword to `break 2`, it will break out of the outer loop, and not just the inner loop.

```

1 $ cat pat.sh
2 for i in {1..5}; do
3   for j in {1..5}; do
4     if [[ "$j" -eq 3 ]]; then
5       break 2;
6     fi
7     echo -n $j
8   done
9   echo
10 done
$ bash pat.sh
12

```

The same works for the `continue` keyword as well.

```

1 $ cat pat.sh
2 for i in {1..5}; do
3   for j in {1..5}; do
4     if [[ "$j" -eq 3 ]]; then
5       continue 2;
6     fi
7     echo -n $j
8   done

```

```

9 echo
10 done
11 $ bash pat.sh
12 1212121212

```

## 9.15 Functions

If the script is too long, it is usually better to split the script into multiple concise functions that does only one task.

There are three ways to define a function in bash:

### **Without the function keyword:**

```

1 abc(){
2   commands
3 }

```

### **function keyword with brackets**

```

1 function abc(){
2   commands
3 }

```

### **function keyword without brackets**

```

1 function abc {
2   commands
3 }

```

### **Example:**

```

1 $ cat functions.sh
2 sayhi(){
3   echo "Hello, $1"
4 }
5
6 sayhi "John"
7 $ bash functions.sh
8 Hello, John

```

## Arguments in Functions

Just like we can use \$1 inside a script to refer to the first argument, we can use \$1 inside a function to refer to the first argument passed to the function.

The arguments passed to the script are not available inside the function, instead the arguments passed to the function are available inside the function.

```

1 $ cat fun.sh
2 fun(){
3     echo $1 $2
4 }
5 echo "Full name:" $1 $2
6 fun "Hello" "$1"
7 $ bash fun.sh John Appleseed
8 Full name: John Appleseed
9 Hello John

```

## Return value of a function

Functions in bash do not return a value, rather they exit with a status code. However, functions can print the value to the standard output, and the caller can capture the output of the function using command substitution.

```

1 $ cat calc.sh
2 add(){
3     echo $($1 + $2)
4 }
5
6 mul(){
7     echo $($1 * $2)
8 }
9
10 select operator in plus multiply ; do
11     read -p "Operand 1: " op1
12     read -p "Operand 2: " op2

```

In this example, we are creating a simple calculator script that takes two operands and an operator, and returns the result. The operator selection is done using a select loop, and the operands are read using `read`. We have modularized the script by creating two functions, `add` and `mul`, that takes two operands and returns the result.

```

13 case "$operator" in
14 plus) answer=$(add $op1 $op2) ;;
15 multiply) answer=$(mul $op1 $op2) ;;
16 *) echo "Invalid option" ;;
17 esac
18 echo "Answer is $answer"
19 done
20 $ bash calc.sh
21 1) plus
22 2) multiply
23 #? 1
24 Operand 1: 5
25 Operand 2: 4
26 Answer is 9
27 #? 2
28 Operand 1: 6
29 Operand 2: 3
30 Answer is 18
31 #?

```

However, sometimes we may want to return from a function as a means to exit the function early. We may also want to signal the success or failure of the function. Both of these can be done using the `return` shell built-in.

In this example, we have created a function that prints the numbers from 0 to the number passed to the function, but if the number passed is negative, the function returns early with a status code of 1. If the number is greater than 9, it only prints till 9, and then returns.

```

$ cat return.sh
fun(){
    if [[ $1 -lt 0 ]]; then
        return 1
    fi
    local i=0
    while [[ $i -lt $1 ]]; do
        if [[ $i -gt 9 ]]; then
            echo
            return
        fi
        echo -n $i
        ((i++))
    done
    echo
}

```

```

16 }
17
18 fun 5
19 echo return value: $?
20 fun 15
21 echo return value: $?
22 fun -5
23 echo return value: $?
24 $ bash return.sh
25 01234
26 return value: 0
27 0123456789
28 return value: 0
29 return value: 1

```

**Remark 9.15.1** If no value is provided to the `return` command, it returns the exit status of the last command executed in the function.

## Local variables in functions

By default, all variables in bash are global, and are available throughout the script. Even variables defined inside a function are global, and are available outside the function. This might cause confusion, as the variable might be modified by the function, and the modification might affect the rest of the script.

To make a variable local to a function, we can use the `local` keyword.

```

1 $ cat local.sh
2 fun(){
3     a=5
4     local b=10
5 }
6
7 fun

```

```

8 | echo "A is $a"
9 | echo "B is $b"
10| $ bash local.sh
11| A is 5
12| B is

```

If a variable is declared using the `declare` keyword, it is global if defined outside a function and local if defined inside a function.

As it is seen in the example, the variable `a` is available outside the function, but the variable `b` is not available outside the function since it is defined using the `local` shell built-in.

## 9.16 Debugging

Sometimes, especially when writing and debugging scripts, we may want to print out each line that the interpreter is executing, so that we can trace the control flow and find out any logical errors.

This can be done by setting the `x` flag of `bash` using `set -x`.

```

1 | $ cat debug.sh
2 | fun(){
3 |   echo $($1 + $2)
4 |
5 |
6 | fun $(fun $(fun 1 2) 3) 4
7 | $ bash debug.sh
8 | 10

```

In the above script, we are calling the `fun` function multiple times, and it is difficult to trace the control flow.

To trace the control flow, we can set the `x` flag using the `set` command.

```

1 | $ cat debug.sh
2 | set -x
3 | fun(){

```

```

4 echo $(( $1 + $2 ))
5 }
6
7 fun $(fun $(fun 1 2) 3) 4
8 $ bash debug.sh
9 +++ fun 1 2
10 +++ echo 3
11 ++ fun 3 3
12 ++ echo 6
13 + fun 6 4
14 + echo 10
15 10

```

Now we can see the control flow of the script, and we can trace the execution of the script. The + is the PS4 prompt, which denotes that the line printed is a trace line and not real output of the script.

**Remark 9.16.1** The trace output of a script is printed to the standard error stream, so that it does not interfere with the standard output of the script.

The PS4 prompt is repeated for each level of the function call, and is incremented by one for each level. This helps us visualize the call stack and order of execution of the script.

## 9.17 Recursion

Just like other programming languages, bash also supports recursion. However, since functions in bash do not return a value, it becomes terse to use recursion in bash by using command substitutions.

```
1 fibo(){
```

```

2 if [[ $1 -le 2 ]]; then
3   echo 1
4 else
5   echo $($((fibo $((1-1)) + $(fibo $((1
6 -2))))) )
7 fi
8 }
9 for i in {1..10}; do
10   fibo $i
11 done

```

In the above example, we are calculating the first ten elements of the fibonacci series using recursion. We need to use **command substitution** to capture the output of the function, and then use **mathematical evaluation** environment to add the two values.

However, most recursive solutions, including this one, are not efficient, and are not recommended for use in bash as they will be very slow. If we time the function with a argument of 20, we see it takes a lot of time.

```

1 $ time fibo 20
2 6765
3
4 real    0m13.640s
5 user    0m10.010s
6 sys     0m3.187s

```

An iterative solution would be much faster and efficient.

```

1 $ cat fibo.sh
2 fibo(){
3   declare -i a=0
4   declare -i b=1
5   for (( i=1; i<=$1; i++ )); do
6     declare -i c=$a+$b
7     a=$b
8     b=$c

```

```
9  done
10 echo $a
11 }
12
13 time fibo 40
14 $ bash fibo.sh
15 102334155
16
17 real    0m0.001s
18 user    0m0.001s
19 sys    0m0.000s
```

The iterative solution is much faster and efficient than the recursive solution.

## 9.18 Shell Arithmetic

We have already seen the mathematical evaluation environment in bash, which is used to evaluate mathematical expressions.

```
1 $ echo $((1 + 2))
2 3
```

However, the mathematical evaluation environment is limited to integer arithmetic, and does not support floating point arithmetic.

### 9.18.1 bc

A more powerful way to do arithmetic in bash is to use the `bc` command.

**Definition 9.18.1 (BC)** `bc` is an arbitrary precision calculator language, and is used to do floating point arithmetic in bash.

```

1 | $ bc <<< "1.5 + 2.5"
2 | 4.0

```

However, bc by default will set the scale to 0, and will truncate the result to an integer.

```

1 | $ bc <<< "10/3"
2 | 3

```

This can be changed by setting the `scale` variable in bc.

```

1 | $ bc <<< "scale=3; 10/3"
2 | 3.333

```

We can also use the `-l` flag to load the math library in bc, which provides more mathematical functions and also sets the scale to 20.

```

1 | $ bc -l <<< "10/3"
2 | 3.333333333333333333333333

```

**7: REPL - Read, Evaluate, Print, Loop** is an interactive interpreter of a language.

bc supports all the basic arithmetic operations, and also supports the `if` statement, `for` loop, and `while` loop and other programming constructs. It is similar to C in syntax, and is a powerful language.

bc can also be started in an interactive mode, which is a REPL.<sup>7</sup>

bc is not just a calculator, rather it is a full fledged programming language, and can be used to write scripts.

```

1 | $ cat factorial.bc
2 | define factorial(n) {
3 |     if(n==1){
4 |         return 1;
5 |     }
6 |     return factorial(n-1) * n;
7 |
8 | $ bc factorial.bc <<< "factorial(5)"
9 | 120

```

Read the man page of bc to know more about the language.

```

1 | $ man bc

```

## 9.18.2 expr

**Definition 9.18.2 (EXPR)** `expr` is a command line utility that is used to evaluate expressions.

As `expr` is a command line utility, it is not able to access the shell variables directly, rather we need to use the `$` symbol to expand the variable with the value before passing the arguments to `expr`.

```
1 $ expr 1 + 2  
2 3
```

The spaces around the operator are necessary, as `expr` is a command line utility, and the shell will split the input by spaces.

The operand needs to be escaped or quoted if it has a special meaning in the shell.

```
1 $ expr 10 \* 3  
2 30  
3 $ expr 10 '*' 3  
4 30  
5 $ ls  
6 $ expr 10 * 3  
7 30  
8 $ touch '+'  
9 $ expr 10 * 3  
10 13
```

The `*` symbol is a special character in the shell, and is used for path expansion. If the folder is empty, then it remains as `*`, but if the directory has files, then `*` will expand to the sorted list of all the files. In this example we show this by creating a file with the name as `+`, thus `10*3` actually expands to `10 + 3` and gives the output of 13.

Like the mathematical environment, `expr` exits a zero exit code if the expression evaluates to a non-zero value, and exits with a non-zero exit code if the expression evaluates to zero.

This inversion is useful in `if` and `while` loops.

```
1 $ expr 5 '>' 6  
2 0  
3 $ echo $?  
4 1
```

```

5 $ expr 5 '<' 6
6 1
7 $ echo $?
8 0

```

`expr` can also be used to match regex patterns. Unlike most other commands, the regex pattern is always anchored to the start of the string.

As seen, the regex pattern is always anchored to the start of the string. The matching is done greedily, and the maximum possible match is taken. `expr` prints the length of the match, and not the match itself.

```

1 $ expr hello : h
2 1
3 $ expr hello : e
4 0
5 $ expr hello : .*e
6 2
7 $ expr hello : .*l
8 4

```

However, if we want to actually print the match instead of the length, we can enclose the regex pattern in escaped parentheses.

```

1 $ expr hello : '\(.*\)'
2 hell

```

Other string operations that `expr` supports are:

- ▶ `length` - Returns the length of the string.
- ▶ `index` - Returns the index of the first occurrence of the substring.
- ▶ `substr` - Returns the substring of the string.

```

1 $ expr length hello
2 5
1 $ expr index hello e
2 2
1 $ expr substr hello 2 3
2 ell

```

The index is 1 based in `expr`, and not 0 based. For the `substr` command, the first argument is the string,

the second argument is the starting index, and the third argument is the length of the substring.

## 9.19 Running arbitrary commands using source, eval and exec

Just like we can use the `source` command to run a script file in the current shell itself, we can also run any arbitrary command directly using `eval` without needing a file.

```
1 $ eval date
2 Wed Jul 31 11:00:42 PM IST 2024
3 $ cat script.sh
4 echo "Hello"
5 $ source script.sh
6 Hello
7 $ eval ./script.sh
8 Hello
```

As it can run any command, it can also run a script file. But the `source` command cannot run a command without a file. However, this can be circumvented by using the `/dev/stdin` file.

```
1 $ source /dev/stdin <<< "echo Hello"
2 Hello
```

**Warning 9.19.1** We should be careful when using the `eval` command, as it can run any arbitrary command in the current shell, and can be a security risk.

### 9.19.1 exec

Similar to the eval command, the exec command can also be used to run arbitrary commands in the same environment without creating a new environment. However, the shell gets replaced by the command being run, and when the command exits, the terminal closes. If the command fails to run then the shell is preserved.

**Exercise 9.19.1** Open a new terminal and run the command `exec sleep 2`. Observe that the terminal closes after 2 seconds.

## 9.20 Getopts

Getopts is a built-in command in bash that is used to parse command line arguments. It is a syntactical sugar that helps us parse command line arguments easily.

The first argument to the getopts command is the string that contains the options that the script accepts. The second argument is the name of the variable that will store the option that is parsed... getopts reads each argument passed to the script one by one, and stores the option in the variable, and the argument to the option in the \$OPTARG variable. It needs to be executed as many times as the number of options passed to the script. As this number is often unknown, it is usually executed in a while loop, since the getopts command returns a non-zero exit code once all the passed options are checked.

```
1 | $ cat optarg.sh  
2 | while getopts ":a:bc:" flag; do
```

```

3   echo "flag -$flag, Argument $OPTARG";
4 done
5 $ bash optarg.sh -a 1 -b -c 2
6 flag -a, Argument 1
7 flag -b, Argument
8 flag -c, Argument 2

```

The colon after the option denotes that the option requires an additional argument. The colon at the start of the string denotes that the script should not print an error message if an invalid option is passed.

### Without the leading colon:

```

1 $ cat optarg.sh
2 while getopts "a:" flag; do
3   echo "flag -$flag, Argument $OPTARG";
4 done
5 $ bash optarg.sh -a 1 -b
6 flag -a, Argument 1
7 optarg: illegal option -- b
8 flag -, Argument
9 $ bash optarg.sh -a
10 optarg: option requires an argument -- a
11 flag -, Argument

```

### With the leading colon:

```

1 $ cat optarg.sh
2 while getopts ":a:" flag; do
3   echo "flag -$flag, Argument $OPTARG";
4 done
5 $ bash optarg.sh -a 1 -b
6 flag -a, Argument 1
7 flag -, Argument b
8 $ bash optarg.sh -a
9 flag :-, Argument a

```

If the option is not passed, the \$OPTARG variable is set to the option itself, and the flag variable is set to :.

If an illegal option is passed, the `flag` variable is set to `?`, and the `$OPTARG` variable is set to the illegal option.

These let the user print a custom error message if an illegal option is passed or if an option that requires an argument is passed without an argument.

### 9.20.1 With case statement

Usually the `getopts` command is used with a `case` statement to execute the code for each option.

```
1 $ cat optarg.sh
2 time="Day"
3 while getopts ":n:mae" opt; do
4     case $opt in
5         n) name=$OPTARG ;;
6         m) time="Morning" ;;
7         a) time="Afternoon" ;;
8         e) time="Evening" ;;
9         \?) echo "Invalid option: $OPTARG" >&2 ;;
10    esac
11 done
12 echo -n "Good $time"
13 if [ -n "$name" ]; then
14     echo ", $name!"
15 else
16     echo "!"
17 fi
18 $ bash optarg
19 Good Day!
20 $ bash optarg -a
21 Good Afternoon!
22 $ bash optarg -e
23 Good Evening!
24 $ bash optarg -m
25 Good Morning!
26 $ bash optarg -an Sayan
27 Good Afternoon, Sayan!
```

```

28 $ bash optarg -mn Sayan
29 Good Morning, Sayan!
30 $ bash optarg -en Sayan
31 Good Evening, Sayan!
32 $ bash optarg -n Sayan
33 Good Day, Sayan!
34 $ bash optarg -n Sayan -a
35 Good Afternoon, Sayan!

```

The error printing can also be suppressed by setting the `OPTERR` shell variable to 0.

## 9.21 Profile and RC files

There are two kinds of bash environments:

- ▶ **Login shell** - A login shell is a shell that is started after a user logs in. It is used to set up the environment for the user.
- ▶ **Non-login shell** - A non-login shell is a shell that is started after the user logs in, and is used to run commands.

When a non-login shell is started, it reads the `~/.bashrc` file, and the `/etc/bash.bashrc` file.<sup>8</sup>

When a login shell is started, along with the **run command** files, it reads the `/etc/profile` file, and then reads the `~/.bash_profile` and `~/.profile` file.

We make most of the configuration changes in the `~/.bashrc` file, as it is read by both login and non-login shells, and is the most common configuration file. Make sure to backup the `~/.bashrc` file before making any changes, as a misconfiguration can cause the shell to not start.

<sup>8</sup>: The `rc` in `bashrc` stands for **run command**. This is a common naming convention in Unix-like systems, where the configuration files are named with the extension `rc`.

## 9.22 Summary

In this chapter, we learned about the control flow constructs in bash, and how to use them to author scripts to automate your work.

**Table 9.3:** Summary of the bash constructs

Construct	Description
<code>if</code>	Execute a block of code based on a condition.
<code>case</code>	Execute a block of code based on a pattern.
<code>for</code>	Iterate over a list of elements.
<code>while</code>	Execute a block of code as long as a condition is true.
<code>until</code>	Execute a block of code as long as a condition is false.
<code>break</code>	Exit the loop immediately.
<code>continue</code>	Skip the rest of the code in the loop and go to the next iteration.
<code>read</code>	Read input from the user.
<code>unset</code>	Unset a variable.
<code>local</code>	Declare a variable as local to a function.
<code>return</code>	Return from a function.
<code>source</code>	Run a script file in the current shell.
<code>eval</code>	Run arbitrary commands in the current shell.
<code>exec</code>	Run arbitrary commands in the current shell.
<code>getopts</code>	Parse command line arguments.

# Stream Editor

# 10

**Stream Editor** (sed) is a powerful text stream editor. It is used to perform basic text transformations on an input stream (a file or input from a pipeline). While in some ways similar to an editor which permits scripted edits (such as ed), sed works by making only one pass over the input(s), and is consequently more efficient. But it is sed's ability to filter text in a pipeline which particularly distinguishes it from other types of editors.

This means that sed can be used in a pipeline of other commands to filter, refine, and transform the data in the stream. This is illustrated in Figure 10.1.

**Remark 10.0.1** sed is short for Stream EDitor.

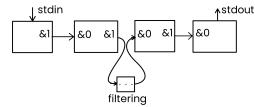


Figure 10.1: Filtering Streams

## 10.1 Basic Usage

Sed needs an input (stream or file) and a script to work. The script is a series of commands that sed will execute on the input. The script can be passed as a command line argument or as a file as well.

For example, we can provide the stream to manipulate as standard input, and provide a single command as the script directly through the command line.

```
1 | $ sed 's/World/Universe/' <<< "Hello World"
2 | Hello Universe
```

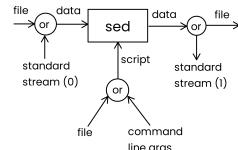


Figure 10.2: The different interfaces to sed

Here we are using the `s` command of sed to perform a find-and-replace style substitution. This is similar to the `%s/regex/string/` present in vim. The first argument can be a regex, but the second argument has to be a string, however it may contain backreferences to the matched pattern.

As sed can work on standard input if no file is provided, it is very useful when paired with other tools; It can transform or filter output of commands.

For example, the default output of date does not zero-pad the date. If we want to do that using sed, we can do it as follows:

This does not add an extra zero if the date is already two digits long. As we have already covered regular expressions in depth, it should be easy to understand the regex used here. We are grouping the regex to use it in the back reference and add an extra zero if the date is a single digit.

```

1 $ date
2 Tue Aug  6 05:00:23 PM IST 2024
3 $ date | sed -E 's/^([[:alpha:]]*[:space:
   :]*)[([[:alpha:]]*)[:space:]]*([([[:digit:]])
   [:space:]):]/\1 0\2 /'
4 Tue Aug 06 05:00:32 PM IST 2024
5 $ date -d "20241225"
6 Wed Dec 25 12:00:00 AM IST 2024
7 $ date -d "20241225" | sed -E 's/^([[:alpha:
   :]*)[([[:space:]]*)[([[:alpha:]]*)[:space:
   :]])*([([[:digit:]]])[:space:]):]/\1 0\2 /'
8 Wed Dec 25 12:00:00 AM IST 2024

```

If the pattern being searched does not match on the input, sed will not make any changes to the input.

## 10.2 Addressing

All of the commands in sed runs on all the lines by default. However, we can specify the lines to run on by providing the optional address field before the command.

```

1 $ cat data
2 hello world
3 hello universe
4 $ sed '1s/hello/hi/' data
5 hi world
6 hello universe

```

The address can be the line number of a particular line, range of line numbers, or a regex pattern to

match. We can also use \$ to match the last line and 1 to match the first line.

```
1 $ seq 10 20 | sed '5,10d'  
2 10  
3 11  
4 12  
5 13  
6 20  
  
1 $ seq 10 20 | sed '/15/,/20/d'  
2 10  
3 11  
4 12  
5 13  
6 14  
  
1 $ seq 10 20 | sed '5d'  
2 10  
3 11  
4 12  
5 13  
6 15  
7 16  
8 17  
9 18  
10 19  
11 20
```

Furthermore, we can also specify the range of addresses to run the command on, using regex as the start, end, or both the ends.

```
1 $ seq 10 20 | sed '/13/, $d'  
2 10  
3 11  
4 12
```

The range usually uses the start and end addresses separated by a comma. But we can also use +n to include n lines starting from the start range.

```
1 $ seq 10 20 | sed '/13/, +5d'
```

```

2| 10
3| 11
4| 12
5| 19
6| 20

```

To match every  $n$ th line, we can use  $\sim n$ .

```

1| $ seq 10 20 | sed '/0~2d'
2| 10
3| 12
4| 14
5| 16
6| 18
7| 20

```

### 10.2.1 Negation

We can negate the address by using the ! operator before the address.

```

1| $ seq 8 5 32 | sed '/1./!d'
2| 13
3| 18

```

In this example we first use the seq command to generate a sequence of numbers from 8 to 32 with a step of 5. Then we use sed to delete all the lines that do not match the pattern 1., printing only the numbers that start with 1.

## 10.3 Commands

### 10.3.1 Command Syntax

The general syntax of the sed command is as follows:

```

1| [:label] [address] command [arguments]

```

### 10.3.2 Available Commands

Sed has a lot of commands that can be used to manipulate the input stream. Here are some of the most commonly used commands:

- ▶ **p** - Print the pattern space.
- ▶ **d** - Delete the pattern space.
- ▶ **s** - Substitute the pattern using regex match with string.  
[address]s/search regex/replace string/[flags]  
]
- ▶ **=** - Print the current line number.
- ▶ **#** - Comment.
- ▶ **i** - Insert text before the current line.
- ▶ **a** - Insert text after the current line.
- ▶ **c** - Change the current line.
- ▶ **y** - Transliterate the characters in the pattern space.
- ▶ **q [exit code]** - Quit the script.

### 10.3.3 Branching and Flow Control

Other than these, there are other control flow commands like **b**, **t**, **:**, which let us create more complex and powerful transformations with state information.

- ▶ **b label** - Branch unconditionally to label.
- ▶ **:label** - Define Label for branching.
- ▶ **n** - Read the next line to the pattern space.
- ▶ **N** - Append the next line to the pattern space.
- ▶ **t label** - Branch to label on successful substitution.
- ▶ **T label** - Branch to label on failed substitution.
- ▶ **w file** - Write the pattern space to file.

- ▶ **r** file - Append the contents of file to the pattern space.
- ▶ **h** - Copy the pattern space to the hold space.
- ▶ **H** - Append the pattern space to the hold space.
- ▶ **g** - Copy the hold space to the pattern space.
- ▶ **G** - Append the hold space to the pattern space.
- ▶ **x** - Exchange the pattern space and the hold space.
- ▶ **d** - Delete the first line of the pattern space.
- ▶ **p** - Print the first line of the pattern space.

More details on these commands can be found in the `man` and `info` pages of `sed`.

#### 10.3.4 Printing

The `p` command prints a line in `sed`. By default the line in pattern space is printed by default. However this can be suppressed using the `-n` flag. Thus when the `p` command is used along with the `-n` flag, only the lines that are explicitly printed are shown.

```
1 | $ seq 10 20 | sed -n '5p'
2 | 14
```

#### 10.3.5 Deleting

Here we are deleting two ranges of lines from the input stream. Multiple commands can be separated by a semi-colon.

The `d` command deletes the line in the pattern space. This is useful if the `-n` flag is not used to suppress the default printing.

```
1 | $ seq 10 20 | sed '1,4d;6,$d'
2 | 14
```

Thus there are two ways of filtering a stream of data as seen in the last two examples:

- ▶ Print only the lines that match a pattern.
- ▶ Delete the lines that do not match a pattern.

**Remark 10.3.1** If we use the `p` command without using the `-n` flag, the matched lines will be printed twice, and the other lines will be printed once.

```
1 $ seq 1 5 | sed '2p'
2 1
3 2
4 2
5 3
6 4
7 5
```

### 10.3.6 Substitution

This is one of the most widely used commands in `sed`. The `substitute` command is used to replace a pattern with a string. The search pattern can be a fixed string, or a regex pattern.

**Remark 10.3.2** `sed` supports two types of regex: Basic and Extended. Perl Compatible Regular Expressions (PCRE) are not supported by `sed`.

Like the other commands, the substitution command can be used with an address to specify the lines to run on, or on all the lines.

```
1 $ seq 8 12 | sed 's/1/one/'
2 8
3 9
4 one0
```

```

5 | one1
6 | one2

```

Observe that although the pattern 1 is present twice in 11, it is only replaced once as the substitution command only replaces the first occurrence of the pattern in the line.

To substitute all the occurrences of the pattern in the line, we can use the g option to the s command.

```

1 | $ seq 8 12 | sed 's/1/one/g'
2 | 8
3 | 9
4 | one0
5 | oneone
6 | one2

```

The g argument stands for global, and it replaces all the occurrences of the pattern in the line.

If we want to replace only the nth occurrence of the pattern, we can mention the number instead of g.

```

1 | $ seq 8 12 | sed 's/1/one/2'
2 | 8
3 | 9
4 | 10
5 | 1one
6 | 12

```

Here the 2 flag replaces the second occurrence of the pattern in the line.

Observe that the lines with only one occurrence of the pattern are not changed at all.

**Remark 10.3.3** If we ask sed to replace the nth occurrence of the pattern, and the pattern does not occur the pattern n times in the line, then the line is not changed at all.

## Case Insensitive Substitution

If we want to perform a case insensitive substitution, we can use the `i` option to the `s` flag.

```
1 $ sed 's/Hello/Hi/i' <<< "hello world"
2 Hi world
```

## Backreferences

Just like in `grep` we can use groups and backreferences in `sed` as well.

```
1 $ sed 's/\([^\,]*\),/\1\n/g' <<< "1,2,3,4,5"
2 1
3 2
4 3
5 4
6 5
```

Here we are matching each of the comma separated values and replacing them with the same value followed by a newline. The group needs to be escaped with a backslash in BRE. However we can use Extended Regular Expressions (ERE) to avoid this by using the `-E` flag.

```
1 $ sed -E 's/([^\,]*),/\1\n/g' <<< "apple,banana
2 ,cherry,donut,eclairs"
3 apple
4 banana
5 cherry
6 donut
7 eclairs
```

We are using groups and backreferences in this case as we are dropping the comma and replacing it with a newline. However, if we want to preserve the entire match and add some text before or after

In these regex, we are matching any character except a comma greedily as long as possible, and replacing it with the same pattern followed by a newline.

it, we can use the & backreference without explicitly grouping the entire match.

```
1 $ sed 's/[^,]*,/&\n/g' <<< "apple,banana,
2     cherry,donut,eclairs"
3 apple,
4 banana,
5 cherry,
6 donut,
7 eclairs
```

**Remark 10.3.4** The & is a backreference to the entire match.

## Uppercase and Lowercase

We can use the \u and \l backreferences to convert the first character of what follows (usually a dynamically fetched backreference) to uppercase or lowercase.

- 1: The uppercasing or lowercasing is done till the end of the string or till the next E backreference.

```
1 $ sed 's/.*/\u&/' <<< hello
2 Hello
```

```
1 $ sed 's/.*/\U&/' <<< hello
2 HELLO
```

To reset the effect of the \u or \l backreference, we can use the \E backreference.

```
1 $ sed 's/(\[^ ]*\) *(\[^ ]*\)/\U\1 \E\u\2/'
2 <<< "hello world"
3 HELLO World
```

Here we are matching two groups of non-space characters (words) using the [^ ]\* regex, separated

by zero or more spaces. Thus the first word is referred to by \1 and the second by \2. Then we are converting the first word to fully uppercase using \U\1, and the first letter of the second word to uppercase, using \u\2.

To reset the effect of the \u backreference, we use the \E backreference.

### 10.3.7 Print Line Numbers

The = command prints the line number of the current line.

```
1 $ seq 7 3 19 | sed '='
2 1
3 2
4 3
5 4
6 5
```

Here we are using the seq command to generate a sequence of numbers from 7 to 19 with a step of 3. Then we are using sed to print the line number of each line. We use the -n flag to suppress the default printing of the line.

#### wc emulation

Since we can print the line number of any line, we can use this to emulate the wc command by printing only the line number of the last line.

```
1 $ seq 7 3 19 | sed -n '$='
2 5
3 $ seq 7 3 19 | wc -l
4 5
```

### 10.3.8 Inserting and Appending Text

The `i` command is used to insert text before the current line. The `a` command is used to append text after the current line.

```
1 $ seq 1 5 | sed '2ihello'
```

```
2 1
3 hello
4 2
5 3
6 4
7 5
```

```
1 $ seq 1 5 | sed '2ahello'
```

```
2 1
3 2
4 hello
5 3
6 4
7 5
```

Here we are using the `seq` command to generate a sequence of numbers from 1 to 5. Then we are using `sed` to insert the text `hello` before the second line.

If we drop the address, the text is inserted before every line.

```
1 $ seq 1 5 | sed 'ihello'
```

```
2 hello
3 1
4 hello
5 2
6 hello
7 3
8 hello
9 4
10 hello
11 5
```

```
1 $ seq 1 5 | sed 'ahello'
```

```
1 1
2 hello
3 2
4 hello
5 3
6 hello
7 4
8 hello
9 5
10 hello
```

We can also insert multiple lines by escaping the newline character.

```
1 $ seq 1 5 | sed '2i\
2 hello\
3 how are you'
4 1
5 hello
6 how are you
7 2
8 3
9 4
10 5
```

### 10.3.9 Changing Lines

The `c` command is used to change the current line. This is sometimes more convenient than substituting the entire line.

Let's say we want to remove all the lines that contains 8 or more digits consecutively, as these may be confidential information, and replace it with "REDACTED".

```
1 $ cat data
2 Hello
3 This is my phone number:
4 9999999998
5 and
```

```

6 this is my aadhaar card number:
7 9999999999999998
8 my bank cvv is
9 123
10 $ sed -E '/[0-9]{8}/cREDACTED' data
11 Hello
12 This is my phone number:
13 REDACTED
14 and
15 this is my aadhaar card number:
16 REDACTED
17 my bank cvv is
18 123

```

Here we are addressing all the lines that contain 8 or more digits consecutively, and replacing them with "REDACTED". The `[0-9]{8}` regex matches any sequence of 8 digits.

### 10.3.10 Transliteration

The `y` command is used to transliterate the characters in the pattern space. This is similar to the `tr` command. However, it is not as powerful as `tr` as it does not support ranges.

```

1 $ sed 'y/aeiou/12345/' <<< "hello world"
2 h2ll4 w4rld

```

Unlike the `tr` command, the `y` command does not support unequal length strings.

## 10.4 Combining Commands

We can combine multiple commands in a single script by separating them with a semicolon.

```

1 $ seq 10 20 | sed '1,4d;6,$d'
2 14

```

Here we are deleting two ranges of lines from the input stream.

However, sometimes we may want to perform multiple commands on the same address range, this can be done by enclosing the commands in braces.

```

1 $ cat data
2 from sys import argv, exit
3
4 def main():
5     if len(argv) != 3+1: # TODO: move magic
6         number to variable
7         print("Incorrect Usage")
8         exit(1)
9
10    kloc,a,b = list(map(float, argv[1:]))
11    # TODO: add adjustment factor
12    return a * (kloc ** b)
13
14
15 if __name__ == "__main__":
16     print(main())
17 $ sed -n '/#/{=;s/^.*# //p}' data
18
19 4
20 TODO: move magic number to variable
21 9
22 TODO: add adjustment factor

```

Here we are addressing the line by matching the regex # which marks the start of a comment. Then on that address we are performing a compound command using braces. The first command (=) prints the line number, and the second command (s ) deletes the # and anything before it.

Here we are performing two actions on the matching lines.

- ▶ Printing the line number of the matching line.
- ▶ Printing only the TODO comment, not the code before it.

This is a handy operation to extract TODO comments from the codebase.

Instead of using a semicolon to separate the commands and addressing each command, we are enclosing them in braces and addressing the group only once.

## 10.5 In-place Editing

As we saw in Figure 10.2, sed can be used to filter the lines of a file as well. We have also seen that in the examples till now. However, we have always printed the output to the terminal. The file remains unchanged.

However, we can use the `-i` flag to edit the file in place.

```

1 $ cat data
2 hello world
3 $ sed -i 's/\b[[:alpha:]]/\u&/g' data
4 $ cat data
5 Hello World
6 $ sed -i 's/\b[[:alpha:]]/\l&/g' data
7 $ cat data
8 hello world

```

In this example we are searching for a line in the configuration file that starts with `LAUNCH_ICBM` and are then running a compound command. The command first tries to substitute `FALSE` with `TRUE`. If it was successful, it then branches to the end of the script to avoid changing it back to `FALSE`.

If the substitution was not successful then it tries to replace `TRUE` with `FALSE`. Thus the command toggles the boolean variable between `TRUE` and `FALSE`. We will cover branching in details later in the chapter.

This is useful when working with large files, as we do not have to write the output to a temporary file and then replace the original file with the temporary file. It is also used widely to modify configuration files in-place without using programming languages.

```

1 $ cat usa.conf
2 LAUNCH_ICBM=FALSE
3 $ sed -i '/^LAUNCH_ICBM/{s/!$&/$&/;tx;s/
4 .TRUE./!$&/$&/}; :x' usa.conf
5 $ cat usa.conf
6 LAUNCH_ICBM=TRUE

```

```

1 $ sed -i '/^LAUNCH_ICBM/{s/FALSE/TRUE/;tx;s/
2   TRUE/FALSE/}; :x' usa.conf
3 $ cat usa.conf
4 LAUNCH_ICBM=FALSE

```

We can also preserve the original file as a backup by providing a suffix to the `-i` flag.

```

1 $ cat data
2 hello
3 $ sed -i.backup 's/Hello/Bye/' data
4 $ cat data
5 bye
6 $ cat data.backup
7 hello

```

## 10.6 Sed Scripts

Instead of listing out the commands as a string in the command line, we can also provide a file of sed commands, called a sed script, where each command is delimited by a newline character instead of the semicolon character.

```

1 $ cat script.sed
2 s/hello/hi
3 $ sed -f script.sed <<< "hello world"
4 hi world

```

Different commands should be present in separate lines in the sed script.

```

1 $ cat script.sed
2 5p
3 10p
4 $ seq 101 120 | sed -n -f script.sed
5 105
6 110

```

### 10.6.1 Order of Execution

2: If the address matches

The sed script is executed for each line in the input stream. The order of checking is as per the order of the stream, and then the order of the script, that is, for each line in the stream from top to bottom, each line of the script is executed<sup>2</sup> from top to bottom.

```

1 $ cat script.sed
2 10p
3 5p
4 $ seq 10 | sed -n -f script.sed
5
6 10

```

Even though the script has the command to print the 10th line before the command to print the 5th line, the output is in the order of the stream.

However, if for a line in the stream, multiple commands in the script match, then the order of the script is followed.

```

1 $ cat script.sed
2 /10/{ 
3   p
4   a Multiple of 10
5 }
6 /5\|10/{ 
7   p
8   a Multiple of 5
9 }
10 $ seq 10 | sed -n -f script.sed
11
12 Multiple of 5
13 10
14 10
15 Multiple of 10
16 Multiple of 5

```

In this example, we are printing the line if it is a multiple of 5 or 10. The case of 5 comes first as the

order of the stream is in increasing order.

However, for the number 10, both the conditions are satisfied, and the order of the script is followed, first the text "Multiple of 10" is printed, and then "Multiple of 5".

**Remark 10.6.1** Observe that the `p` command prints the line, and the `a` command appends text after the line. Even though the first `a` is present before the second `p`, still both the `p` are printed first and then the `a` are printed. This is because the `p` command prints the line immediately, but the `a` command appends the text to the hold space, and prints it only when the next line is read.

## 10.6.2 Shebang

If we are running a `sed` script directly from the shell without specifying the interpreter, we need to add the shebang in the first line of the script.

```
1 |#!/bin/sed -f
```

or

```
1 |#!/usr/bin/sed -f
```

This indicates to the shell that the interpreter to use to run this script is `sed`.

The `-f` flag is required to make `sed` read the script from the script file.

**Remark 10.6.2** The bash shell simply invokes

the command in the script's shebang along with all of the arguments and appends the path of the script file as the last argument of the parameter list.

Note that the script file has to be made executable using the `chmod` command. This is a security feature to prevent accidental execution of random untrusted scripts.

Note that we have also specified the `-n` flag to suppress the default printing of the line.

```

1 $ cat script.sed
2 #!/bin/sed -nf
3 a Hello
4 bx
5 a World
6 :x
7 a Universe
8 $ chmod u+x script.sed
9 $ ./script.sed <<< ""
10 Hello
11 Universe

```

## 10.7 Branching and Flow Control

Sed supports branching and flow control commands to create more complex transformations using loop and if-else conditions. However, sed does not support syntax like `if-else`, `for`, or `while` loops, thus to perform these control flow operations, we have to use the branching commands.

This is similar to how high level languages are compiled to assembly language **GOTO** commands, and then to machine code.

### 10.7.1 Labels

Labels are used to mark a point in the script to branch to. They themselves do not perform any action, but are used as a reference point for branching.

```

1 $ cat script.sed
2 :label1
3 5p
4 :label2
5 10p
6 $ seq 10 | sed -n -f script.sed
7 5
8 10

```

## 10.7.2 Branching

Now that we have defined labels in our script, we can use branching to instruct sed to move the control flow to any arbitrary label.

**Remark 10.7.1** The label can be defined before, or after the branching call

There are two kinds of branching in sed,

- ▶ **Unconditional Branching** - the branching does not depend on success or failure of previous command.
- ▶ **Conditional Branching** - the branching is conditioned on the success or failure of the substitution done before the branching call.

### Unconditional Branching

The `b` command is used to branch unconditionally to a label. This will always branch to the label and skip any command after it.

```

1 $ cat script.sed
2 a Hello
3 bx
4 a World

```

```

5 :x
6 a Universe
7 $ sed -nf script.sed <<< "test"
8 Hello
9 Universe

```

In this example the command `a Hello` is executed, and then the `bx` command is executed. So, the command `a World` is skipped, and the control flow is moved to the label `x`, and the command `a Universe` is executed.

If the label is after the branch, it can be used to skip certain parts of the script. However, if the label is before the branch, it can be used to loop over the same part of the script multiple times. But if done using an unconditional branch, it will result in an infinite loop.

```

1 $ cat script.sed
2 :x
3 p
4 bx
5 $ sed -nf script.sed <<< "hello"
6 hello
7 hello
8 hello
:
:
```

The `bx` command branches to the label `x` unconditionally, and thus the command `p` is executed multiple times. To stop the infinite loop, we can press `Ctrl+C`.

However, we can use a regex address to limit the number of times the loop is executed.

```

1 $ cat script.sed
2 :x
3 s/\b[a-z]/\u&/
4 /\b[a-z]/bx

```

```
5 $ cat data
6 this is a long line with lots of words, we
    want to capitalize each first letter of a
    word.
7 $ sed -f script.sed data
8 This Is A Long Line With Lots Of Words, We
    Want To Capitalize Each First Letter Of A
    Word.
```

In this example, we are first declaring a branch label x. Then we are using the substitution command to capitalize the first letter of the first word, observe that we are not using the g flag to capitalize all the words. Then we are using a regex address to check if there are any more words in the line. Only if there are more words, we branch to the label x. Thus we are using a unconditional branch but we are conditionally branching to the label using the regex address.

This can be simplified using the t command.

## Conditional Branching

The t command is used to branch to a label if the previous substitution was successful.

```
1 $ cat script.sed
2 :x
3 s/\b[a-z]/\u&/
4 tx
5 $ cat data
6 this is a long line with lots of words, we
    want to capitalize each first letter of a
    word.
7 $ sed -f script.sed data
8 This Is A Long Line With Lots Of Words, We
    Want To Capitalize Each First Letter Of A
    Word.
```

Here we are using the `t` command to branch to the label `x` if the substitution was successful. This will terminate once all the words are capitalized as in the next iteration the substitution will fail.

Thus this implementation will run one iteration more than the previous implementation since it terminates when the substitution fails inside the iteration.

### 10.7.3 Appending Lines

We can use the `N` command to append the next line to the pattern space. This will keep the same iteration cycle, but simply append the next line to the current line. The next line is delimited from the current line by a newline character. After processing the pattern space, the iteration cycle directly moves to the line after the appended line.

A line read normally by sed does not include the newline character at the end of the line, however, if the next line is appended to pattern space using `N`, then the newline character is included in the pattern space.

```

1 $ cat script.sed
2 1~2N
3 =
4 $ seq 10 | sed -n -f script.sed
5 2
6 4
7 6
8 8
9 10

```

In this example, we are appending the next line to the pattern space for every odd line. Then we are printing the line number of the current line.

Since we are appending the next line to the pattern space, the line number is printed for the even lines.

```
1 $ cat script.sed
2 N
3 N
4 s/\n;/;g
5 $ seq 10 | sed -n -f script.sed
6 1;2;3
7 4;5;6
8 7;8;9
9 10
```

In this example, we are appending the next two lines to the pattern space. Then we are replacing the newline characters with a semicolon. Observe the g flag in the substitution command, this is used to replace all the newline characters in the pattern space.

Since sed does not re-process the appended lines in the next cycle, the next line starts from 4.

**Remark 10.7.2** The N command terminates the script on the last line in GNU sed, but is undefined in the POSIX standard.

**Exercise 10.7.1** Try the previous example without the g flag in the substitution command. Predict the output before running it, observe and justify the output after running it. How many lines are printed? How many iteration cycles does sed go through in this?

### 10.7.4 Join Multiline Strings

If we have a file where long lines are hard-wrapped using a \\ character at the end of the line, we can use the N command to join the lines and recreate the real file.

```

1 $ cat data
2 Hello, this is a ver\
3 y long line, but I h\
4 ave broken it into s\
5 everal lines of fixe\
6 d width and marked f\
7 orced breaks with a \
8 backwards slash befo\
9 re the newline chara\
10 cter.
11 Real newlines do not\
12 have the backslash \
13 though.
14 Like this.
15 $ cat script.sed
16 :x
17 /\$/{
18   N
19   s/\n/
20   bx
21 }
22 $ sed -f script.sed data
23 Hello, this is a very long line, but I have
      broken it into several lines of fixed
      width and marked forced breaks with a
      backwards slash before the newline
      character.
24 Real newlines do not have the backslash though
      .
25 Like this.

```

In this script, we first are defining a label x. Then we are checking if the line ends with a \\ character. If it does, then we are appending the next line to the

pattern space. Then we are substituting the \\ and the newline character with an empty string, this joins the word-wrapped lines back. Finally, we are branching back to the label x to check if the current pattern space still has a \\ at the end.

There are other ways of performing the same action, such as using t and P.

**Exercise 10.7.2** Try to implement the same script using the t and P commands.

### 10.7.5 If-Else

We can also use **labels** and **branching** to emulate the if-else syntax.

```

1 $ cat script.sed
2 /(1|3|5|7|9)/{$
3   s/$/ is an odd number/
4   bx
5 }
6 s/$/ is an even number/
7 :x
8 $ seq 10 | sed -Ef script.sed
9 1 is an odd number
10 2 is an even number
11 3 is an odd number
12 4 is an even number
13 5 is an odd number
14 6 is an even number
15 7 is an odd number
16 8 is an even number
17 9 is an odd number
18 10 is an even number

```

We need to use Extended Regular Expressions (ERE) if we are using the parenthesis grouping and pipe-or syntax of ERE without escaping them. This is why we have used the -E flag. We can also drop it and escape the parentheses and pipes.

The substitution adds the text "is an odd/even number" at the end of each number. Substituting \$ simply appends to the end of the line.

Here, the first command group is addressed explicitly for only those lines which end in odd digits. If

the number is odd, it will enter this block, append the addage that the number is odd, and then branch to the end of the script and move on to next line.

However, if the number is not odd, it will not enter this command group, and thus not execute the branch. Thus it will execute the second substitution.

### 10.7.6 If-elif-else

Similarly we can construct a **if-elif-else** ladder.

```
1 $ cat script.sed
2 /(1|3|5|7|9)$/{           # If the number is odd
3     s/$/ is an odd number/
4     bx
5 }
6 /0$/{
7     s/$/ is zero/
8     bx
9 }
10 s/$/ is an even number/
11 :x
12 $ seq 0 5 | sed -Ef script.sed
13 0 is zero
14 1 is an odd number
15 2 is an even number
16 3 is an odd number
17 4 is an even number
18 5 is an odd number
```

## 11.1 What is AWK?

Awk is programming language meant for processing structured data in rows and columns.

The name **AWK** comes from the initials of its creators:

- ▶ Alfred Aho
- ▶ Peter Weinberger
- ▶ Brian Kernighan

**Remark 11.1.1** Alfred Aho was the author of *egrep* and thus **awk** works with Extended Regular Expressions directly, and not Basic Regular Expressions.

The most popular distribution of awk is **gawk** - GNU AWK, developed by the GNU team and distributed under the GPL.

Kernighan also released his version **nawk** - New AWK, that is used by BSD systems to avoid GPL.

AWK was the only scripting language in user space in early days along with the bourne shell. It inspired the creation of the **Perl** language which also has powerful text processing tools.

The awk language is a data-driven language, that is, it processes data line by line. It is a procedural language, but it is not object-oriented. It has a very simple syntax, and it is very powerful for text processing. For each line that is read, awk applies a set



Figure 11.1: Alfred Aho



Figure 11.2: Peter Weinberger



Figure 11.3: Brian Kernighan

of rules to the line and executes the actions that are associated with the rules.

## 11.2 Basic Syntax

The basic syntax of an awk program is a series of pattern action pairs, written as:

```
1 | pattern { action }
```

The entire input is split into records, and each record is split into fields. By default records are split using newlines, so each record is one line of input, but it can be changed.

Each record is tested against the pattern, and if the pattern matches, the action is executed.

Thus, an awk program has an implicit loop that reads each record, tests each record against the patterns, and executes the actions if they match.

Here we are checking for numbers in each line and if the number ends with a 5 then we are printing that it is a multiple of 5. Similarly if it ends with a 0, then we print that it is a multiple of 5 and 10. Observe how this is achieved using regex alternation and multiple pattern-action pairs.

```
1 | $ cat script.awk
2 | /5$|0$/ {
3 |     print $0 " is a multiple of 5"
4 | }
5 |
6 | /0$/ {
7 |     print $0 " is a multiple of 10"
8 | }
9 | $ seq 20 | awk -f script.awk
10 | 5 is a multiple of 5
11 | 10 is a multiple of 5
12 | 10 is a multiple of 10
13 | 15 is a multiple of 5
14 | 20 is a multiple of 5
15 | 20 is a multiple of 10
```

If the pattern is omitted, the action is executed for every record.

```
1 $ cat script.awk
2 {
3     print "Line: " $0
4 }
5 $ seq 5 | awk -f script.awk
6 Line: 1
7 Line: 2
8 Line: 3
9 Line: 4
10 Line: 5
```

If the action is omitted, the default action is to print the record.

```
1 $ seq 50 | awk '/5$/' 
2 5
3 15
4 25
5 35
6 45
```

### 11.2.1 Special Conditions

- ▶ **BEGIN** - This pattern is executed before the first record is read.
- ▶ **END** - This pattern is executed after the last record is read.

```
1 $ cat script.awk
2 BEGIN{
3     print "Starting the script"
4 }
5 {
6     print "Line: " $0
7 }
8 END{
9     print "Ending the script"
10 }
11 $ seq 5 | awk -f script.awk
12 Starting the script
```

```

13 Line: 1
14 Line: 2
15 Line: 3
16 Line: 4
17 Line: 5
18 Ending the script

```

## 11.2.2 Ranges

The condition can also be a range of records, matched by their record number or by regular expression patterns. Both the ends are inclusive.

Here we are printing from the line starting with a capital T and till the line that ends with a full stop.

```

1 $ cat data.txt
2 Hello
3 This is the start
4 of a long sentence
5 spanning multiple
6 lines in the file.
7 Goodbye
8 $ awk '/^T/,/\.$/' 
9 This is the start
10 of a long sentence
11 spanning multiple
12 lines in the file.

```

The record number can be matched using the `NR` variable. It is a variable built-in awk and it is automatically updated everytime a new record is read.

```

1 $ seq 10 | awk 'NR == 3, NR == 7'
2 3
3 4
4 5
5 6
6 7

```

### 11.2.3 Fields

Awk splits each record into fields. By default the fields are split by any kind of whitespace, including spaces and tabs. However, this can be changed to split by some other delimiter or even regular expression. This is useful for different structured files such as CSV and quoted CSV files.

The fields are numbered from 1 and can be accessed using the `$i` syntax.

```
1 $ echo "Hello World" | awk '{print $1}'
2 Hello
```

The variable `$0` variable contains the entire record.

We can also match a regular expression pattern against a specific field as well using the `~` syntax.

```
1 $ echo -e "Hello World\nhello universe\nthis
2     is hello" | awk '$1 ~ /[hH]ello/'
3 hello
```

### 11.2.4 Number of Fields and Number of Records

The number of fields in a record can be accessed using the `NF` variable. It is a built-in variable in awk.

```
1 $ echo "Hello World" | awk '{print NF}'
2 2
```

The number of records can be accessed using the `NR` variable. It is a built-in variable in awk.

```
1 $ seq 5 10 | awk 'END{print NR}'
2 6
```

Here we are printing the number of fields in each line. The number of fields may change from record to record, the variable `NR` will also be updated accordingly automatically.

Here we are printing the `NR` variable which is updated everytime a new record is read. Thus the `NR` variable stores the value  $i$  when it is reading the  $i^{th}$  record. However, we are not printing it for each record, rather we are printing it only after all the records are read, in the `END` block. The `NR` variable is not wiped after reading the last record, thus it still contains the record number of the last record, which is same as the total number of records as we count from 1.

### 11.2.5 Splitting Records by custom delimiter

By default awk splits records by whitespace. However, we can change this to split by some other delimiter. There are two variables that control this behavior:

- ▶ **FS** - Field Separator - This is the regular expression that is used to split the fields in a record.
- ▶ **FPAT** - Field Pattern - This is the regular expression that is used to match the fields in a record.

The **FS** variable can be set in the BEGIN block, or it can be set from the command line using the **-F** option. The **FPAT** variable can be set in the BEGIN block.

```
1 | $ echo "Hello,World" | awk '{print $1}'
2 | Hello,World
3 | $ echo "Hello,World" | awk -F, '{print $1}'
4 | Hello
```

Here we can see that by default the record is split by whitespace, and thus the entire line is considered as a single field. However, when we set the **FS** variable to a comma, then the record is split by the comma and we get the first field as **Hello**.

The **FS** variable can also be explicitly set inside the script, if for example we do not have access to the command line flags.

```
1 | $ echo "Hello,World" | awk 'BEGIN{FS=", "}{'
2 |   print $1'
2 | Hello
```

The **FS** variable can also be set from the command line by using the **-v** flag.

```

1 $ echo "Hello,World" | awk -v FS="," '{print
2     $1}'
2 Hello

```

Similarly, the `FPAT` variable can be set in the `BEGIN` block or sent from the commandline using the `-v` flag.

The `FPAT` variable is used to match the fields in a record. We can use regular expressions in this to match any particular pattern of fields if the field separator is not fixed.

```

1 $ cat script.awk
2 BEGIN{
3     FPAT="[a-zA-Z]+"
4 }
5 {
6     for(i=1; i<=NF;i++){
7         printf $i " "
8     }
9     print ""
10 }
11 $ cat data.txt
12 Hello1This2is3a4file^with:lots'of'delimiters
13 however,the-fields=are+always/alphabets
14 so%we#can@simply!use*FPAT-variable?to"match
15 each]field.
16 $ awk -f script.awk data.txt
17 Hello This is a file with lots of delimiters
18 however the fields are always alphabets
19 so we can simply use FPAT variable to match
20 each field

```

In this example we have set the `FPAT` variable to match any alphabets. This is useful when the field separator is not fixed, but the fields themselves have a fixed pattern.

Then, for each line, we are printing all the fields in that line separated by a space.

## 11.3 Awk Scripts

Even though awk is great for one-lines, it is also great for writing scripts. The scripts can be written in a file and then executed using the `awk -f` flag. This is similar to how sed scripts also work, this chapter assumes familiarity with the previous chapter.

```

1 $ cat script.awk
2 BEGIN{
3     print "Starting the script"
4     FS=", "
5 }
6 {
7     print "First Field: " $1
8 }
9 END{
10    print "Ending the script"
11 }
12 $ cat data.csv
13 Hello,World
14 Goodbye,Universe
15 $ awk -f script.awk data.csv
16 Starting the script
17 First Field: Hello
18 First Field: Goodbye
19 Ending the script

```

The entire contents of the script file is executed as an awk script. The BEGIN and END blocks are executed before and after the records are read respectively. All other pattern blocks<sup>1</sup> are executed for each record which match the pattern.

1: Here we are using a block without any pattern, so it matches all lines.

### 11.3.1 Command Line Arguments

Awk scripts can also take command line arguments. These arguments can be accessed using the ARGV

array. The ARGC variable stores the number of arguments.

```

1 $ cat script.awk
2 BEGIN{
3     print ARGC
4     for(arg in ARGV){
5         print ARGV[arg]
6     }
7 }
8 $ awk -f script.awk hello how "are you"
9 4
10 awk
11 hello
12 how
13 are you

```

In awk, the first argument is the name of the program itself, and the rest of the arguments are the arguments passed to the program. This is consistent with C, which also uses variables `argc` and `argv` as parameter names in the `main()` function.

`awk` will assume that the first command line argument is the script and the rest of the command line arguments are the files to work on.

However, if we do not have any pattern-action pairs in the script and we do not have an `END` block as well, then `awk` not try to open the arguments as files. This is why we did not get any error in the last example. If we add a empty block after the `begin`, we will encounter the file not found error.

```

1 $ cat script.awk
2 BEGIN{
3     print ARGC
4     for(arg in ARGV){
5         print ARGV[arg]
6     }
7 }
8 {}

```

```

9 $ awk -f script.awk hello how "are you"
10 4
11 awk
12 hello
13 how
14 are you
15 awk: script.awk:5: fatal: cannot open file '
      hello' for reading: No such file or
      directory

```

Note that the command block has no action in it, so it will not perform any action, but because it is present awk will still expect to open the file.

If we pass filenames as arguments instead to the same script, then the error subsides, even if nothing is printed from the files.

```

1 $ cat script.awk
2 BEGIN{
3     print ARGV
4     for(arg in ARGV){
5         print ARGV[arg]
6     }
7 }
8 {}
9 $ awk -f script.awk /etc/fstab ~/.bashrc
10 3
11 awk
12 /etc/fstab
13 /home/sayan/.bashrc

```

**Remark 11.3.1** Note that having a action-less condition like `NR==5` will print the records by default which match the pattern, whereas having a condition-less action like `{ print $1 }` will execute that action for all lines. We can also have a pattern-less and action-less block as seen in the previous example `{}`. In this case, it will do

nothing for all the lines. However awk will still expect one or more files to perform this NO-OP on, and providing a command line argument that is not a valid file path will throw an error as we saw. Further, even if the block is empty, awk will still read each line of the file, this can be verified by running the NO-OP on an infinite file like /dev/zero or /dev/urandom and observing that the command never stops.

```
1 | $ awk '{}' /dev/zero
```

Press `Ctrl+C` to stop the command, as it may eat up your system resources since there are no newlines in `/dev/zero`, hence the record read in memory keeps on increasing.

This gives rise to two small NO-OP scripts that can be used to read the entire file.

```
1 | $ awk '{}' /etc/fstab # does nothing
2 | $ awk '1' /etc/fstab # prints the entire file
```

Thus `awk 1` can be used as a `cat` replacement.

Similarly `awk 0` or `awk {}` can be used as a `test -r` replacement to test if a file exists and is readable.

**Exercise 11.3.1** Use the aforementioned tricks to write a script that first tests if the file path mentioned as the first argument of the script (`$1`) exists or not (using awk), and if it does exist, then print its contents (using awk).

**Solution:**

```
1 | awk 0 "$1" && awk 1 "$1"
```

### 11.3.2 Shebang

Awk scripts can also be made executable by adding a shebang line at the top of the script. The current shell will then execute the script using the awk interpreter. Throughout this book we shall be using the gawk interpreter for running awk scripts. Similar to sed scripts, we have to mention the -f flag in the shebang.

```

1 $ cat script.awk
2 #!/usr/bin/gawk -f
3 BEGIN{
4     print "hello"
5 }
6 $ chmod +x script.awk
7 $ ./script.awk
8 hello

```

To run the script the file has to be made executable using the `chmod +x` command.

## 11.4 Variables

Just like any other programming language, awk has support for variables to store data. There are three types of variables in awk, they are:

- ▶ Scalar - A variable that holds a single value of any valid data type.
- ▶ Indexed Array - An array that is indexed by numbers.
- ▶ Associative Array - An array that is indexed by strings.

Variables in awk are dynamically typed, that is, the type of the variable is determined by the value assigned to it. Further, the type of the variable

can change during the execution of the program. The type is automatically determined by the value assigned to it or the operation performed on it.<sup>2</sup>

A variable is initially of the type `unassigned`, however as awk implicitly sets the default value of variables, that is equivalent to 0 and empty string `"."`.

2: GNU Awk also has support for regex-typed variables, which are variables that can store a regular expression to match.

```
1 $ awk 'BEGIN { print (a == "" && a == 0 ? "a
      is untyped" : "a has a type!"); print
      typeof(a) }'
2 a is untyped
3 unassigned
```

The type of a variable can be checked using the `typeof()` function.

```
1 $ awk 'BEGIN { a=5; b="hello"; c=5.5; print
      typeof(a); print typeof(b); print typeof(c
      ) }'
2 number
3 string
4 number
```

### 11.4.1 Loose Typing and Numeric Strings

In a dynamically typed language, it is easy to infer when a variable is string. But what about number inputs? Input, by design, are always strings in awk. However, if the string can be converted to a number, then it is treated as a number. These kinds of strings are called numeric strings.

When a numeric string is operated with a number, it is converted to a number in that expression. However, when it is operated with a string, it is converted to a string in that expression.

```
1 $ echo "9" | awk '{ print $1 < 10 }'
2 1
```

```
3 | $ echo "9" | awk '{ print $1 < "10" }'
4 | 0
```

In the above example, the first expression is comparing a numeric string with a number, so the string is converted to a number, and we perform a numeric comparison, that is, is the number 9 less than the number 10? And thus the output is 1, which means true in awk.

In the second expression, we are comparing a numeric string with a string, so the numeric string is converted to a string, and we perform a string comparison, that is, does the string "9" come lexicographically before the string "10"? The answer is no since in string comparison we compare letter by letter, and '9' is not less than '1'.

Only user input can be a numeric string, a string that looks like a number present in the script itself can never be treated like a number.

```
1 | $ echo "hello 123" | awk '{ print typeof($1),
                                typeof($2), typeof(1), typeof("1"),
                                typeof("one") }'
2 | string strnum number string string
```

The above example demonstrates that the input fields can either be string or strnum (numeric string) if they look like a number, but never number. Similarly, the numbers in the script are always numbers, and never numeric strings, and the string in the script are always strings, never numeric strings.

You can read more about variable typing in awk [here](#).

### 11.4.2 User-defined Variables

To define a variable, we simply assign a value to it. The variable is created when it is assigned a value. As awk is dynamically typed, the type of the variable is determined by the value assigned to it. A variable can be assigned a value using the = operator. This can be done in the following places:

- ▶ In the BEGIN block.
- ▶ In the pattern-action pairs.
- ▶ In the END block.
- ▶ In the command line using the -v flag.
- ▶ In the command line without the -v flag.

If the variable is assigned as a command line argument without the -v flag, the time of assignment of the variable depends on its order in the command line arguments.

If the assignment is done after the first file and before the second file, the variable will be unset for all the records of the first file and set to the value for the second file.

```

1 $ echo hello > hello
2 $ echo world > world
3 $ awk '{ print n $0 }' n=1 hello n=2 world
4 1hello
5 2world

```

The above example demonstrates that the variable n is set to 1 for the first file and 2 for the second file following the order of the command line arguments.

Similarly, if the variable is not defined at all, then it is treated as an empty string.

```

1 $ echo hello > hello
2 $ echo world > world

```

```

3 $ awk '{ print n $0 }' hello n=2 world
4 hello
5 2world

```

If we are using the `-v` flag to set the variable, it has to mandatorily be done before all files and before the awk script or script file also. However it can still be overwritten in subsequent arguments.

```

1 $ awk -v n=1 '{ print n $0 }' hello n=2 world
2 1hello
3 2world

```

The variables can also be declared in the `BEGIN`, `END`, or the pattern-action pairs.

A variable declared inside the blocks will be available for use in all subsequent blocks of that iteration and also of all blocks of the subsequent iterations. Unfamiliarity with this may lead to logical errors in the script.

```

1 $ cat script.awk
2 #!/bin/gawk -f
3
4 $0 % 2 == 0 {
5     mult2 = 1
6     print $0 " is a multiple of 2"
7 }
8
9 $0 % 5 == 0 {
10    mult5 = 1
11    print $0 " is a multiple of 5"
12 }
13
14 mult2 == 1 && mult5 == 1 {
15     print $0 " is a multiple of 10"
16 }
17 $ seq 10 | awk -f script.awk
18 2 is a multiple of 2
19 4 is a multiple of 2
20 5 is a multiple of 5

```

```

21 5 is a multiple of 10
22 6 is a multiple of 2
23 6 is a multiple of 10
24 7 is a multiple of 10
25 8 is a multiple of 2
26 8 is a multiple of 10
27 9 is a multiple of 10
28 10 is a multiple of 2
29 10 is a multiple of 5
30 10 is a multiple of 10

```

As it is seen in the above example, the variables `mult2` and `mult5` are declared in the pattern-action pairs, and they are used in the subsequent pattern-action pair. However, even for next iterations, the variables are not reset, and they are still available for use in the next iteration. This causes logical error which prints that all numbers are multiples of 10. This is because the variables are available for use in all subsequent blocks of that iteration and also of all blocks of the subsequent iterations.

To fix this issue, we have to reset the variables at the end of the iteration.

```

1 $ cat script.awk
2 #!/bin/gawk -f
3
4 $0 % 2 == 0 {
5     mult2 = 1
6     print $0 " is a multiple of 2"
7 }
8
9 $0 % 5 == 0 {
10    mult5 = 1
11    print $0 " is a multiple of 5"
12 }
13
14 mult2 == 1 && mult5 == 1 {
15     print $0 " is a multiple of 10"
16 }

```

```

17 {
18     mult2 = 0
19     mult5 = 0
20 }
21 $ seq 10 | awk -f script.awk
22 2 is a multiple of 2
23 4 is a multiple of 2
24 5 is a multiple of 5
25 6 is a multiple of 2
26 8 is a multiple of 2
27 10 is a multiple of 2
28 10 is a multiple of 5
29 10 is a multiple of 10

```

Due to this, the `BEGIN` block is the best place to declare variables that are used throughout the script. However, if your declaration simply sets a numeric variable to 0 or a string variable to an empty string, then it is not necessary to declare the variable at all, as `awk` implicitly sets undefined variables to 0 or empty string when they are used for the first time.

```

1 $ cat script.awk
2 #!/bin/gawk -f
3 BEGIN{
4     prod = 1
5 }
6 {
7     prod *= $0
8     sum += $0
9 }
10 END{
11     print "product:", prod
12     print "sum:", sum
13 }
14 $ seq 5 | awk -f script.awk
15 product: 120
16 sum: 15

```

In the above example, we are calculating the prod-

uct and sum of the numbers in the records. We need to initialize the variables `prod` and `sum` in the `BEGIN` block, as they are used throughout the script. However, as the `sum` variable is initialized to 0 we do not need to declare it explicitly. The `prod` variable is initialized to 1 as it is used in a multiplication operation, and multiplying by 0 will always result in 0, so it has to be explicitly declared in the `BEGIN` block.

### 11.4.3 Built-in Variables

There are a lot of built-in variables in awk, they either let us configure how awk works, or gives us access to the state of the program.

#### Variables that control awk

There are a lot of variables that control how awk works by configuring behaviours and edge-cases. We will not cover all of them, but some of the important ones.

A full list can be found in the [manual](#).

- ▶ `FS` - Field Separator - This is the regular expression that is used to split the fields in a record. It cannot match a null string.
- ▶ `FPAT` - Field Pattern - This is the regular expression that is used to match the fields in a record. It can match a null string.
- ▶ `FIELDWIDTHS` - Field Width - This is the width of the fields in a record.
- ▶ `CONVFMT` - Conversion Format - This is the format used to convert numbers to strings.
- ▶ `OFORMAT` - Output Format - This is the format used to print numbers.

- ▶ ORS - Output Record Separator - This is the string that is printed after each record.
- ▶ OFS - Output Field Separator - This is the string that is printed between each field.
- ▶ RS - Record Separator - This is the string that is used to split input records.

`FIELDWIDTHS` and `FPAT` are gawk extensions and are not available in all versions of awk.

Let us play around with these variables to understand them better.

**FS:**

```

1 $ cat script.awk
2 #!/bin/gawk -f
3
4 BEGIN{
5   FS=" , "
6 }
7
8 NR != 1{
9   print $1
10  marks+=$2
11  students++
12 }
13 END{
14   print "Average Marks:", marks/students
15 }
16 $ cat data.csv
17 Name,Marks
18 Adam,62
19 Bob,76
20 Carla,67
21 Delphi,89
22 Eve,51
23 $ awk -f script.awk data.csv
24 Adam
25 Bob
26 Carla
27 Delphi

```

```
28 | Eve
29 | Average Marks: 69
```

In this example we are splitting the records by a comma, and then printing the name of the student and calculating the average marks of the students.

As the file is a normal CSV file, we can use the `FS` to split the records by the comma delimiter.

If we do not set the `FS` variable, then the records are split by whitespace, and the entire line is considered as a single field. Try it out with names with multiple words and see how the fields are split.

We also use two user-defined variables `marks` and `students` to calculate the average marks of the students.

We can also use the `NR` variable to get the number of records instead of creating and maintaining a separate `students` variable. However note that we also have a header row, so we have to exclude that from the count.

```
1 | $ cat script.awk
2 | #!/bin/gawk -f
3 |
4 | BEGIN{
5 |   FS=","
6 | }
7 |
8 | NR != 1{
9 |   print $1
10 |   marks+=$2
11 | }
12 | END{
13 |   print "Average Marks:", marks/(NR-1)
14 | }
15 | $ cat data.csv
16 | Name,Marks
17 | Adam,62
```

```

18 Bob,76
19 Carla,67
20 Delphi,89
21 Eve,51
22 $ awk -f script.awk data.csv
23 Adam
24 Bob
25 Carla
26 Delphi
27 Eve
28 Average Marks: 69

```

We will cover NR and other built-in variables in the next subsection.

#### FPAT:

The FPAT variable is used to match the fields in a record. It can match a null string. If the field separator is not fixed, then we can use the FPAT variable to match the fields in a record. If we set the FPAT variable, then the FS variable is ignored, and vice-versa.

If we want to replicate the previous example using the FPAT variable, we can do it as follows:

```

1 $ cat script.awk
2 #!/bin/gawk -f
3
4 BEGIN{
5   FPAT="[^,]*"
6 }
7
8 NR != 1{
9   print $1
10  marks+=$2
11 }
12 END{
13  print "Average Marks:", marks/(NR-1)
14 }
15 $ cat data.csv

```

```

16 Name,Marks
17 Adam,62
18 Bob,76
19 Carla,67
20 Delphi,89
21 Eve,51
22 $ awk -f script.awk data.csv
23 Adam
24 Bob
25 Carla
26 Delphi
27 Eve
28 Average Marks: 69

```

As we can see the output remains the same, but the `FPAT` variable is used to match the fields in the record.

The regex `[^,]*` matches any character except a comma, and it matches zero or more of these characters. We will discuss more about regex subsequent sections.

### **FIELDWIDTHS:**

Sometimes the fields in a record are of fixed width, and they are not separated by any delimiter. In that case we can use the `FIELDWIDTHS` variable to specify the width of each field.

Let us create a file that contains all the student roll numbers.

```

1 $ for term in {21..24}{1..3}; do for i in
   {1..10000}; do printf "%s%06d\n" $term "$i
   " ; done ; done > student-data

```

It is left as an exercise to the reader to explain the above command and how it is creating all the roll numbers.

3: In this case we already know the answer as we ourselves created the data using those parameters, but assume that the data was not created by us and we want to extract these details from the data.

Then we can use the `FIELDWIDTHS` variable to split the records by the field width and extract the unique years and terms.<sup>3</sup>

```
$ cat script.awk
#!/bin/gawk -f

BEGIN{
    FIELDWIDTHS="2 1 1 6"
}

{
    years[$1]++
    terms[$3]++
}

END{
    print "Years:"
    for(year in years){
        print year
    }
    print "Terms:"
    for(term in terms){
        print term
    }
}
$ awk -f script.awk student-data
Years:
21
22
23
24
Terms:
1
2
3
```

4: Like a dictionary in python

As it is visible, the awk script uses the fixed width fields to split the record and then extract them. We then use an associative array<sup>4</sup> to store the count of each. In the END we use a for loop to iterate over all the keys and print them.

We have not covered either associative arrays or loops till now, we will cover them in the later sections.

### OFMT:

The `OFMT` variable is used to set the output format of numbers.

This is a C-style format string, and it is used to convert numbers to strings during printing.

The default value is "`\%.6g`".

```

1 $ cat script.awk
2 #!/bin/gawk -f
3
4 {
5     sum+=$1
6 }
7 END{
8     print "Average:", sum/NR
9     OFMT="%d"
10    print "Average:", sum/NR
11 }
12 $ seq 10 | awk -f script.awk
13 Average: 5.5
14 Average: 5

```

In this example, we are calculating the average of the numbers in the records. The input were the numbers from 1 to 10. The actual average is 5.5 as seen in the first line of output where the `OFMT` is not modified yet. After that, the `OFMT` is set to "`\%d`", which is the format string for integers, and then the average is printed again, but this time as an integer, so the fractional part gets truncated.

### ORS:

The `ORS` variable is used to set the output record separator. By default, the `ORS` variable is set to the newline character "`\n`".

Consider the following example where we set the FS and ORS variable to extract a comma separated list of user names from the /etc/passwd file.

```

1 $ cat script.awk
2 #!/bin/gawk -f
3
4 BEGIN{
5   FS=":"
6   ORS=","
7 }
8 {
9   print $1
10}
11$ awk -f script.awk /etc/passwd
root,bin,daemon,mail,ftp,http,nobody,dbus,
systemd-coredump,systemd-network,systemd-
oom,systemd-journal-remote,systemd-resolve
,systemd-timesync,tss,uuidd,_talkd,avahi,
named,cups,dnsmasq,git,nm-openconnect,nm-
openvpn,ntp,openvpn,polktd,rpc,rpcuser,
rtkit,saned,sddm,usbmux,sayan,brltty,
gluster,qemu,colord,dhcpcd,fwupd,geoclue,
libvirt-qemu,mysql,monero,mpd,nbd,passim,
postgres,redis,tor,unbound,metabase,test,
flatpak,
```

Here first we are setting the FS to be the colon symbol, which is required as the /etc/passwd file is colon separated. Then we print only the username field, which is the first field. But we set the ORS to be a comma, so that the output is a comma separated list of user names.

This is a useful use of awk, and usually if you are want to accomplish this task you will usually do this using a single line from the command line directly, without making an elaborate script file.

```
1 | awk -F: -vORS=, '{print $1}' /etc/passwd
```

One of the most common use of changing ORS and

RS is to convert from **CRLF** to **LF** or vice-versa.

**Exercise 11.4.1** Write an awk script that will convert a file from **CRLF** to **LF**.

## Variables that convey information

There are many variables that awk sets on its own to convey information about the state of the program. They are also updated by awk whenever applicable.

- ▶ **NF** - Number of Fields - This is the number of fields in the current record. It starts from 1.
- ▶ **NR** - Number of Records - This is the number of records that have been read so far. Also referred to as the Record Number. It starts from 1.
- ▶ **FNR** - File Number of Records - This is the number of records that have been read so far in the current file. It starts from 1 and resets to 1 when a new file is read.
- ▶ **FILENAME** - File Name - This is the name of the current file being read.
- ▶ **RSTART** - Record Start - This is the index of the start of the matched string in the last call to the `match()` function.
- ▶ **RLENGTH** - Record Length - This is the length of the matched string in the last call to the `match()` function.
- ▶ **ENVIRON** - Environment Variables - This is an associative array that contains the environment variables that awk receives.
- ▶ **ARGC** - Count of Command Line Arguments - It is the number of command line arguments passed from the shell. It also counts the name of the executable as the first argument.

- ▶ ARGV - The indexed array that contains all the command line arguments passed from the shell when running the awk program. The first argument is the name of the program itself (most times awk or gawk) and the rest arguments follow. This does not include the awk script itself, or the flags passed to the command.

### FNR and NR:

We have already seen that **NR** denotes the record number in that iteration and is automatically updated by awk every next record.

The **FNR** variable is similar to the **NR** variable; it also stores the record number. However, the **FNR** variable resets from 1 for each new file, whereas the **NR** variable keeps on increasing for all the records read from all the files.

If we are iterating over only a single file, then the **FNR** and **NR** variables will have the same value. However, if we are iterating over multiple files, then the **FNR** variable will reset to 1 for each new file, whereas the **NR** variable will keep on increasing for all the records read from all the files. So the value of **FNR** will always be less than or equal to the value of **NR**.

In this example we are using the `<(command)` syntax to pass the output of the command as a file to awk. We are doing this so we can pass multiple files to awk. Here we can observe that the **FNR** variable resets to 1 for each new file, whereas the **NR** variable keeps on increasing for all the records read from all the files.

```

1 $ awk '{ print FNR, NR }' <(seq 5) <(seq 5)
2 1 1
3 2 2
4 3 3
5 4 4
6 5 5
7 1 6
8 2 7
9 3 8
10 4 9
11 5 10

```

The variables are equal for the first file, but they differ for the subsequent files. This can be exploited as a condition to check if we are in the first file or not.

For example, if we want to print the lines common to both the files, we can do it as follows:

```
1 $ awk ' FNR==NR{ lines[$0]=1 } FNR!=NR &&
        lines[$0]' <(seq 0 3 100) <(seq 0 5 100)
2 0
3 15
4 30
5 45
6 60
7 75
8 90
```

#### NF:

The `NF` variable stores the number of fields in the current record. This can not only be used to see the number of fields in the record, but also to access the last field of the record.

```
1 $ cat data.txt
2 hello,world
3 hello,world,how,are,you
4 $ awk -F, '{ print NF, $NF }' data.txt
5 2 world
6 5 you
```

Here we are using the `NF` variable to print the number of fields in the record and the `$NF` to print the last field in each record.

#### FILENAME:

The `FILENAME` variable stores the name of the current file being read. This can be used in multitude of ways, like to print the file name in the output, or to check if we are in a specific file or not.

Here we are using `FNR==NR` to check if we are in the first file or not. For the first file we are simply storing all the lines in a dictionary. For the second file, which we check using `FNR!=NR`, we are checking if the line is already present in the dictionary. This means that the line has occurred in file one, as well as in the second file, then we print the file. Here the files are multiples of 3 and multiples of 5, so the lines common are the multiples of 15.

Consider the following examples:

```

1 $ cat data1.txt
2 This is the first file
3 There are two lines in this file
4 $ cat data2.txt
5 This is the
6 second file
7 and it has
8 four lines.
9 $ awk '{ print FILENAME ":" FNR ":" $0 }'
   data*
10 data1.txt:1:This is the first file
11 data1.txt:2:There are two lines in this file
12 data2.txt:1:This is the
13 data2.txt:2:second file
14 data2.txt:3:and it has
15 data2.txt:4:four lines.

```

This example demonstrates how we can use the FILENAME variable to print the file name in the output. This is useful if there are a lot of files and we want to prefix their lines with the filename and line number to find which file has a particular line.

Or, if you want to print the filename at the start of the file, we can use FNR to test if it is the first line, then print the filename. We also have to explicitly print all lines.

```

1 $ cat data1.txt
2 This is the first file
3 There are two lines in this file
4 $ cat data2.txt
5 This is the
6 second file
7 and it has
8 four lines.
9 $ awk ' FNR==1 { print "==" FILENAME "==" } 1'
   data*
10 ==data1.txt==
11 This is the first file

```

```
12 There are two lines in this file
13 ==data2.txt==
14 This is the
15 second file
16 and it has
17 four lines.
```

Or, if we want to print how many lines each files have:

```
1 $ cat data1.txt
2 This is the first file
3 There are two lines in this file
4 $ cat data2.txt
5 This is the
6 second file
7 and it has
8 four lines.
9 $ awk -f script.awk data*
10 data2.txt:4
11 data1.txt:2
```

Here we use a dictionary where the key is the FILENAME and the value keeps increasing for each line that exists in the file. This creates a dictionary with the number of lines in each file as we are iterating over all lines of all files.

We can then print the dictionary at the end to get the number of lines in each file. Note that order of the files is not guaranteed, as awk does not guarantee the order of the keys in the dictionary like python before version 3.7.

**Exercise 11.4.2** Run `wc -l data*` and observe the output, can you mimic this output using awk? We have already seen how to store and print the number of lines for each file, can you also print the total sum?

Hint: Use NR.

### ENVIRON:

The ENVIRON variable is an associative array that contains the environment variables that awk receives.

Thus we can also use the shell's environment variables in awk.

```
1 $ export a=5
2 $ awk 'BEGIN{ print ENVIRON["a"] }'
3 5
```

Note that the name of the variable has to be quoted since we are accessing a dictionary and the name of the variable is merely a key in the ENVIRON dictionary. If we do not quote the variable's name, awk will think we are referring to a variable in awk named a and resolve it to an empty string (or its value if defined) and try to access that key in the ENVIRON dictionary, which might not exist or be a different output than expected.

### ARGC and ARGV

The ARGC and ARGV variables, like in C, store the number of commandline arguments passed to the awk program and the string array of each argument. Any flags passed to awk is not part of the ARGV array. The awk script or the awk script file path is also not a part of the array or the count of ARGC.

```
1 $ cat script.awk
2 BEGIN {
3     for(arg in ARGV){
4         print(ARGV[arg])
5     }
6 }
7 $ awk -f script.awk a b c
8 awk
```

```

9 a
10 b
11 c

```

Not unlike C, the first argument is the name of the executable, and the rest are the arguments passed to the executable.

We can directly iterate over the array using a for-each loop, and print the arguments passed to the awk program.

We can also use a C-style for loop to iterate over the array using the ARGC count.

```

1 $ cat script.awk
2 BEGIN {
3     for(i=0; i<ARGC; i++){
4         print(ARGV[i])
5     }
6 }
7 $ awk -f script.awk a b c
8 awk
9 a
10 b
11 c

```

We will discuss the RSTART and RLENGTH variables when we discuss the `match()` function.

## 11.5 Functions

If we have a set of code that performs a specific operation and might be re-used several times, we can move it into its own function and call the function.

Awk has support for functions, and we can define our own functions in awk.

When a function is called, expressions that create the function's actual parameters are evaluated completely before the call is performed.

### 11.5.1 User-defined Functions

Definitions of functions can appear anywhere between the rules of an awk program. There is no need to put the definition of a function before all uses of the function. This is because awk reads the entire program before starting to execute any of it.

Syntax of a function definition is as follows:

```
1 | function name(parameter-list) {  
2 |     statements  
3 | }
```

The name of a function has to follow the same rules as that of a variable and cannot be a name that is already used by a variable, array, or built-in functions.

`parameter-list` is an optional list of the function's arguments and local variable names, separated by commas. When the function is called, the argument names are used to hold the argument values given in the call.

A function cannot have two parameters with the same name, nor may it have a parameter with the same name as the function itself.

The parameter list does not enforce the number of arguments passed to the function, and the function can be called with fewer arguments as well. Any argument that is not passed is treated as an empty string in string contexts and zero in numeric contexts.

```

1 $ cat script.awk
2 function test(a,b,c,d,e,f){
3     print a,b,c,d,e,f
4 }
5
6 BEGIN{
7     test(1,2,3)
8     test(1,2,3,4,5)
9     test(1,2,3,4,5,6)
10}
11$ awk -f script.awk
12 1 2 3
13 1 2 3 4 5
14 1 2 3 4 5 6

```

However providing more arguments than the function expects throws an error.

All the built-in variables are also available inside the function.

```

1 $ cat script.awk
2 function foo(){
3     print NR ":" $0
4 }
5
6 /5/{
7     foo()
8 }
9 $ seq 50 | awk -f script.awk
10 5: 5
11 15: 15
12 25: 25
13 35: 35
14 45: 45
15 50: 50

```

During execution of the function body, the arguments and local variable values hide, or shadow, any variables of the same names used in the rest of the program. The shadowed variables are not accessible in the function definition, because there is

no way to name them while their names have been taken away for the arguments and local variables. All other variables used in the awk program can be referenced or set normally in the function's body.

The arguments and local variables last only as long as the function body is executing. Once the body finishes, you can once again access the variables that were shadowed while the function was running.

All the built-in functions return a value to their caller. User-defined functions can do so also, using the `return` statement.

```

1 $ cat script.awk
2 function sqr(x){
3     return x*x
4 }
5
6 {
7     print sqr($0)
8 }
9 $ seq 5 | awk -f script.awk
10 1
11 4
12 9
13 16
14 25

```

Here the function takes a parameter  $x$  and returns its square  $x^2$ . This is then used by the `print` function to print it to the screen.

A function can also be called recursively.

```

1 $ cat script.awk
2 function factorial(x){
3     if(x==1) return 1
4     return x * factorial(x-1)
5 }
6
7 {

```

```

8   print factorial($0)
9 }
10 $ seq 5 | awk -f script.awk
11 1
12 2
13 6
14 24
15 120

```

## 11.5.2 Pass by Value and Pass by Reference

In awk, all arguments that are not array are passed by value. This means that the variable itself is not passed to the function when calling a function, rather its value is copied and passed to the function. If the function changes the value inside the function it has no effect on the variable used by the caller to pass the value to the function and that variable's value remains unchanged.

```

1 $ cat script.awk
2 function alter(x){
3     print "X inside " x
4     x=x+1
5     print "X inside " x
6 }
7
8 BEGIN{
9     x=5
10    print "X outside " x
11    alter(x)
12    print "X outside " x
13 }
14 $ awk -f script.awk
15 X outside 5
16 X inside 5
17 X inside 6
18 X outside 5

```

However, if we pass an array to a function, then the array is passed by reference. This means that the function can change the array and the changes will be reflected in the array used by the caller to pass the array to the function.

```
1 $ cat script.awk
2 function alter(l){
3     l[0] = 0
4 }
5
6 BEGIN{
7     l[0]=1
8     print l[0]
9     alter(l)
10    print l[0]
11 }
12 $ awk -f script.awk
13 1
14 0
```

### 11.5.3 Built-in Functions

There are multiple built-in functions in awk which make the programming of scripts easier.

Each built-in function accepts a certain number of arguments. In some cases, arguments can be omitted. The defaults for omitted arguments vary from function to function and are described under the individual functions. In some awk implementations, extra arguments given to built-in functions are ignored. However, in gawk, it is a fatal error to give extra arguments to a built-in function.

## Numeric Functions

There are multiple in-built functions that help perform numeric manipulation to ease the development process. They are:

- ▶ `atan2(y,x)` - Returns the arc tangent of  $y/x$  in radians.
- ▶ `cos(x)` - Return the cosine of  $x$  in radians.
- ▶ `exp(x)` - Returns the exponential of  $x = e^x$ .
- ▶ `int(x)` - Return the nearest integer to  $x$ , located between  $x$  and zero and truncated toward zero. For example, `int(3)` is 3, `int(3.9)` is 3, `int(-3.9)` is -3, and `int(-3)` is -3 as well.
- ▶ `log(x)` - Return the natural logarithm of  $x$ , if  $x$  is positive; otherwise, return NaN ("not a number") on IEEE 754 systems. Additionally, gawk prints a warning message when  $x$  is negative.
- ▶ `rand()` - Return a random number. The values of `rand()` are uniformly distributed between zero and one. The value could be zero but is never one.
- ▶ `sin(x)` - Return the sine of  $x$ , with  $x$  in radians.
- ▶ `sqrt(x)` - Return the positive square root of  $x$ . gawk prints a warning message if  $x$  is negative. Thus, `sqrt(4)` is 2.
- ▶ `srand(x)` - Set the starting point, or seed, for generating random numbers to the value  $x$ . Each seed value leads to a particular sequence of random numbers. Thus, if the seed is set to the same value a second time, the same sequence of random numbers is produced again. If no argument is provided then the current date and time is used to set the seed value. This generates unpredictable pseudo-random numbers. It returns the previous seed value.

## String Functions

`gawk` understands locales and does all string processing in terms of characters, not bytes. This distinction is particularly important to understand for locales where one character may be represented by multiple bytes. Thus, for example, `length()` returns the number of characters in a string, and not the number of bytes used to represent those characters. Similarly, `index()` works with character indices, and not byte indices.

String functions are used to manipulate strings in `awk`. They are:

- ▶ `asort(source[,dest[,how]])` - `gawk` sorts the values of `source` and replaces the indices of the sorted values of `source` with sequential integers starting with one. If the optional array `dest` is specified, then `source` is duplicated into `dest`. `dest` is then sorted, leaving the indices of `source` unchanged. This means that `asort` can also work on associative arrays, however, after sorting the keys are no longer preserved, rather it is converted into an indexed array.
- ▶ `asorti(source[,dest,[how]])` - Similar to `asort`, however, it sorts the indices of an associative array instead of the values and produces a indexed array.
- ▶ `gensub(regexp, replacement, how[, target])`
  - This is a `gawk` extension that allows for general substitution. Search the target string `target` for matches of the regular expression `regexp`. If `how` is a string beginning with 'g' or 'G' (short for "global"), then replace all matches of `regexp` with `replacement`. Otherwise, treat `how` as a number indicating which match of `regexp` to replace. Treat numeric

values less than one as if they were one. If no target is supplied, use \$0. It **returns** the modified string as the result of the function. The original target string is **not changed**. The returned value is always a string, even if the original target was a number or a regexp value. `gensub()` is a general substitution function. Its purpose is to provide more features than the standard `sub()` and `gsub()` functions.

- ▶ `gsub(regex,replacement[, target])` - This function searches the target string `target` for all occurrences of the regular expression `regex`, and replaces them with `replacement`. It returns the number of substitutions made. The target string `target` is **changed**. If `target` is omitted then the `$0` is searched and altered.
- ▶ `index(in,find)` - This function searches the string `in` for the first occurrence of the string `find`, and returns the position in `in` where that occurrence begins. If `find` is not found in `in`, then `index()` returns zero. The indices are 1-indexed.
- ▶ `length([string])` - This function returns the length of the string `string`. If it is omitted then the entire `$0` is used.
- ▶ `match(string,regexp[, array])` - This function searches the string `string` for the longest, leftmost substring matched by the regular expression `regexp`. It returns the position in `string` where the matched substring begins, or zero if no match is found. It also sets the `RSTART` and `RLENGTH` variables. `RSTART` is set to the index of the start of the match, and `RLENGTH` is set to the length of the match.
- ▶ `patsplit(string, array[ ,fieldpat[, seps]])` - This function splits the string `string` into fields in the array `array` using the field pattern `fieldpat`. It uses `fieldpat` to match fields. It

is a regular expression. If `fieldpat` is omitted then the value of `FPAT` is used instead. It returns the number of fields created. The original string `string` is not changed. The separator strings are stored in the `seps` array. The `seps` array is indexed by the field number. The value of each element is the separator string that was found before the corresponding field. The `seps` array has one more element than the `array` array.

- ▶ `split(string, array[, fieldsep[, seps]])` - This function splits the string `string` into fields in the array `array`, using the field separator `fieldsep`. It returns the number of fields created. The original string `s` is not changed. If `fieldsep` is omitted, then `FS` is used.
- ▶ `sprintf(fmt,expr-list)` - This function returns the string resulting from formatting the arguments according to the format string `fmt`. The format string is the same as that used by the `printf()` function. The result is not printed.
- ▶ `strtonum(str)` - This function converts the string `str` to a number. It returns the numeric value represented by `str`. If `str` starts with `0` then the number is treated as an octal number. If the string starts with `0x` or `0X` then the number is treated as a hexadecimal.
- ▶ `sub(regexp, replacement[, target])` - This function searches the target string `target` for the leftmost longest occurrence of the regular expression `regexp`, and replaces it with `replacement`. It returns the number of substitutions made (zero or one). The target string `t` is changed. If `target` is omitted then the `$0` is used.
- ▶ `substr(string, start[, length])` - This function returns the substring of `string` starting at position `start` and of length `length`. If `length`

is omitted then the substring till the end of the string is taken.

- ▶ **tolower(string)** - This function returns a copy of the string `string` with all the upper-case characters converted to lower-case.
- ▶ **toupper(string)** - This function returns a copy of the string `string` with all the lower-case characters converted to upper-case.

Let us explore some of these functions with examples.

### **asort and asorti**

```

1 $ cat script.awk
2 BEGIN{
3     a["one"] = "banana"
4     a["three"] = "cherry"
5     a["two"] = "apple"
6     asort(a)
7     for(i in a){
8         print i, a[i]
9     }
10 }
11 $ awk -f script.awk
12 1 apple
13 2 banana
14 3 cherry

```

However, if we use `asorti` instead of `asort`, then the keys are sorted instead of the values.

```

1 $ cat script.awk
2 BEGIN{
3     a["one"] = "banana"
4     a["three"] = "cherry"
5     a["two"] = "apple"
6     asorti(a)
7     for(i in a){
8         print i, a[i]
9     }
10 }

```

```

11 $ awk -f script.awk
12 1 one
13 2 three
14 3 two

gensub

1 $ cat script.awk
2 BEGIN{
3     s = "Hello world"
4     print gensub(/l+/, "_", "g", s)
5 }
6 $ awk -f script.awk
7 He_o wor_d

```

Similarly you can try out with gsub and sub.

### **index and length**

```

1 $ cat script.awk
2 BEGIN{
3     s = "This is a string"
4     print index(s, "is")
5     print length(s)
6 }
7 $ awk -f script.awk
8 3
9 16

```

### **match**

```

1 $ cat script.awk
2 BEGIN{
3     s = "Hello World"
4     print match(s,/l+/,a)
5     print RSTART
6     print RLENGTH
7     for(i in a){
8         print i ":" a[i]
9     }
10 }
11 $ awk -f script.awk
12 3

```

```

13| 3
14| 2
15| 0start: 3
16| 0length: 2
17| 0: ll

```

Using the `a` array we can get the matched string, the start of the match and the length of the match.

### split and patsplit

```

1 $ cat script.awk
2 BEGIN{
3     s = "this,is,a,csv,file"
4     split(s,a,",",seps)
5     for(i in a){
6         print i, a[i]
7     }
8     for(i in seps){
9         print i, seps[i]
10    }
11 }
12 $ awk -f script.awk
13 1 this
14 2 is
15 3 a
16 4 csv
17 5 file
18 1 ,
19 2 ,
20 3 ,
21 4 ,

```

However, the field separator may not always be a fixed string. In that case, we can use the `patsplit` function.

```

1 $ cat script.awk
2 BEGIN{
3     s = "This!is.a,line^with*different&
4       delimiters"
5     patsplit(s,a,/[^A-Za-z]+/,seps)

```

```

5   for(i in a){
6     print i, a[i]
7   }
8   for(i in seps){
9     print i, seps[i]
10  }
11 }
12 $ awk -f script.awk
13 1 This
14 2 is
15 3 a
16 4 line
17 5 with
18 6 different
19 7 delimiters
20 0
21 1 !
22 2 .
23 3 ,
24 4 ^
25 5 *
26 6 &
27 7

```

In this case, the `seps` array is useful as it stores the separators that were found before each field. Note that the first and last element of the `seps` array are empty strings, as there is no separator before the first field and after the last field. Also note that the `seps` array starts from 0 and not 1.

### `substr`

```

1 $ cat script.awk
2 BEGIN{
3   s = "Hello World"
4   print substr(s,1,4)
5   print substr(s,7)
6 }
7 $ awk -f script.awk
8 Hell
9 World

```

If we do not specify the length, then the substring is taken till the end of the string.

### **tolower and toupper**

```

1 $ cat script.awk
2 BEGIN{
3     s = "A string"
4     print tolower(s)
5     print toupper(s)
6 }
7 $ awk -f script.awk
8 a string
9 A STRING

```

There are other types of in-built functions as well, such as [Input Output Functions](#), [Time Functions](#), and [Bitwise Functions](#) which we will not explore in depth here, but students are encouraged to go through the manual for those as well.

The `system()` function is of particular interest as it allows us to run shell commands from within awk. We will cover this in a later section.

## **11.6 Arrays**

Arrays are a collection of elements, where each element is identified by an index or a key. In awk, arrays are associative, which means that the index can be a string or a number, or indexed, where the key is numeric.

Arrays in awk are created on the fly, and there is no need to declare them before using them. The first time an array element is referenced, the array is created.

```

1 $ cat script.awk
2 BEGIN{

```

```

3 arr[0] = "hello"
4 arr[1] = "world"
5 print arr[0] arr[1]
6 delete arr[0]
7 print arr[0] arr[1]
8 delete arr
9 print arr[0] arr[1]
10 }
11 $ awk -f script.awk
12 helloworld
13 world

```

Elements of the arrays can be accessed using the index or key. The individual elements can be deleted using the `delete` function. We can also delete the entire array by passing the array name to the `delete` keyword.

### 11.6.1 Associative Arrays

Associative arrays are arrays where the index is a string or a number. The index can be any string or number, and the elements are stored in the array in no particular order. In awk, unlike modern python, the order of the keys in the array is not guaranteed. The keys, even if they are numbers, are stored as strings and need not be contiguous.

```

1 $ cat script.awk
2 BEGIN{
3     arr["name"] = "John"
4     arr["age"] = 24
5     for(i in arr){
6         print i, arr[i]
7     }
8     print "age" in arr
9     print "gender" in arr
10 }
11 $ awk -f script.awk

```

```

12 age 24
13 name John
14 1
15 0

```

Observe a few points here:

- ▶ The keys are stored as strings, even if they are numbers.
- ▶ The keys are not stored in any particular order.
- ▶ We can check if a key exists in the array using the `in` keyword.
- ▶ We can iterate over the keys of the array using the `for` loop.

### 11.6.2 Multidimensional Arrays

POSIX awk supports multidimensional arrays by converting the indices to strings and concatenating them with a separator. The separator is the value of the `SUBSEP` variable. The default value of `SUBSEP` is the string "\034", which is the ASCII value of the record separator. After the value is stored there is no way to retrieve the original indices, so it is not possible to iterate over the keys of the array.

We can check if an index exists in the array using the `in` keyword.

```

1 $ cat script.awk
2 BEGIN{
3     arr[0,0] = 5
4     print arr[0,0]
5     print (0,0) in arr
6     print (1,1) in arr
7 }
8 $ awk -f script.awk
9 5
10 1
11 0

```

However, this is not really a multidimensional array. It stores all the values in a single associative single dimensional array.

However, gawk supports true multidimensional arrays. Elements of a subarray are referred to by their own indices enclosed in square brackets, just like the elements of the main array.

```

1 $ cat script.awk
2 BEGIN {
3     arr[0][0] = 5
4     print arr[0][0]
5     print 0 in arr
6     print 0 in arr[0]
7     print 1 in arr
8 }
9 $ awk -f script.awk
10 5
11 1
12 1
13 0

```

Similarly we can use nested for-loops to iterate over the elements of the array.

```

1 $ cat script.awk
2 BEGIN{
3     arr[0][0] = 0
4     arr[0][1] = 1
5     arr[1][0] = 2
6     arr[1][1] = 3
7
8     for(i in arr){
9         for(j in arr[i]){
10             printf arr[i][j] " "
11         }
12         printf "\n"
13     }
14 }
15 $ awk -f script.awk
16 0 1

```

17 | 2 3

## 11.7 Regular Expressions

As we have seen before, regular expressions are patterns that describe sets of strings. They are used to search for patterns in text.

Awk by default uses extended regular expressions.

A regular expression can be used as a pattern by enclosing it in slashes. Then the regular expression is tested against the entire text of each record. (Normally, it only needs to match some part of the text in order to succeed.)

For example, the below script tests if the record contains the string bash and prints the first field if it does. Here we are iterating over the records of the file /etc/passwd and printing those users who can login using the bash shell.

```
1 $ awk '/bash/ { print $1 }' /etc/passwd
2 root:x:0:0::/root:/bin/bash
3 sayan:x:1000:1001:Sayan:/home/sayan:/bin/bash
4 postgres:x:946:946:PostgreSQL
```

We can also match a regex on any arbitrary field or string using the ~ operator. Similarly, we can negate the match using the !~ operator.

```
1 $ cat script.awk
2 $5 ~ /[A-Z]/{
3   print $1, $5
4 }
5 $ awk -F: -f script.awk /etc/passwd
6 nobody Kernel Overflow User
7 dbus System Message Bus
8 systemd-coredump systemd Core Dumper
9 systemd-network systemd Network Management
```

```

10| systemd-oom systemd Userspace OOM Killer
11| systemd-journal-remote systemd Journal Remote
12| systemd-resolve systemd Resolver
13| systemd-timesync systemd Time Synchronization
14| _talkd User for legacy talkd server
15| avahi Avahi mDNS/DNS-SD daemon
16| named BIND DNS Server
17| nm-openconnect NetworkManager OpenConnect
18| nm-openvpn NetworkManager OpenVPN
19| ntp Network Time Protocol
20| openvpn OpenVPN
21| polkitd PolicyKit daemon
22| rpc Rpcbind Daemon
23| rpcuser RPC Service User
24| rtkit RealtimeKit
25| saned SANE daemon user
26| sddm SDDM Greeter Account
27| sayan Sayan
28| brltty Braille Device Daemon
29| gluster GlusterFS daemons
30| qemu QEMU user
31| colord Color management daemon
32| fwupd Firmware update daemon
33| geoclue Geoinformation service
34| libvirt-qemu Libvirt QEMU user
35| mysql MariaDB
36| nbd Network Block Device
37| passim Local Caching Server
38| postgres PostgreSQL user
39| redis Redis in-memory data structure store
40| metabase Metabase user
41| flatpak Flatpak system helper

```

Here we are matching the regex [A-Z] on the fifth field of the /etc/passwd file and printing the first and fifth fields if the regex matches. This prints all those records which have a capitalized name for the user.

### 11.7.1 Matching Empty Regex

An empty regex matches the invisible empty string at the start and end of the string and between each character. It can be visualized by using the `gsub()` function and replacing the empty strings with a character.

```

1 $ cat script.awk
2 BEGIN{
3     a = "hello"
4     gsub(//, "X", a)
5     print a
6 }
7 $ awk -f script.awk
8 XhXeXlXlXoX

```

### 11.7.2 Character Classes

Awk supports character classes in regular expressions. A character class is a set of characters enclosed in square brackets. It matches any one of the characters in the set. The character classes discussed in the earlier chapters are consistent with the character classes supported by awk.

**Remark 11.7.1** Awk also supports collating symbols and equivalence classes, however gawk does not support these.

Awk always matches the longest leftmost string that satisfies the regular expression. This is called the maximal match rule. For simple match/no-match tests, this is not so important. But when doing text matching and substitutions with the `match()`, `sub()`, `gsub()`, and `gensub()` functions, it is very important.

Understanding this principle is also important for regexp-based record and field splitting.

### 11.7.3 Regex Constant vs String Constant

The righthand side of a `~` or `!~` operator need not be a regexp constant (i.e., a string of characters between slashes). It may be any expression. The expression is evaluated and converted to a string if necessary; the contents of the string are then used as the regexp. A regexp computed in this way is called a dynamic regexp or a computed regexp.

```
1 $ cat script.awk
2 BEGIN{
3     r = "[A-Z].*apple"
4 }
5 $0 ~ r
6 $ awk -f script.awk /usr/share/dict/words
7 Balmawapple
8 John-apple
```

Here we are matching the regex stored in the variable `r` against the records of the file `/usr/share/dict/words`.

**Remark 11.7.2** When using the `~` and `!~` operators, be aware that there is a difference between a regexp constant enclosed in slashes and a string constant enclosed in double quotes. If you are going to use a string constant, you have to understand that the string is, in essence, scanned twice: the first time when awk reads your program, and the second time when it goes to match the string on the lefthand side of the operator with the pattern on the right. This is true of any string-valued expression (such as `r`, shown in

the previous example), not just string constants.

This matters when we are using escape sequences in the string constant. If using string constants then we have to escape the character twice, once for the string and once for the regex.

For example, `/*` is a regexp constant for a literal `*`. Only one backslash is needed. To do the same thing with a string, you have to type `"\\*"`. The first backslash escapes the second one so that the string actually contains the two characters `\` and `*`.

However, we should always use regex constants over string constants because:

- ▶ String constants are more complicated to write and more difficult to read. Using regexp constants makes your programs less error-prone. Not understanding the difference between the two kinds of constants is a common source of errors.
- ▶ It is more efficient to use regexp constants. awk can note that you have supplied a regexp and store it internally in a form that makes pattern matching more efficient. When using a string constant, awk must first convert the string into this internal form and then perform the pattern matching.
- ▶ Using regexp constants is better form; it shows clearly that you intend a regexp match.

#### 11.7.4 Ignoring Case

Usually regex is case sensitive, and there is no way to make a regex case insensitive from the regex itself. grep tackles this by providing the `-i` flag.

In awk, we can perform case insensitive matching in three ways:

- ▶ By matching both the uppercase and lowercase characters. [wW] matches both w and W.
- ▶ By using the `tolower()` or `toupper()` functions. We can convert the data to lowercase and have the regex in lowercase as well.
- ▶ Using the `IGNORECASE` variable. If set to any non-zero value, awk will ignore case when matching regex.

```

1 $ cat script.awk
2 BEGIN {
3     x = "Hello"
4     if (x ~ /hello/) {
5         print "first check"
6     }
7     IGNORECASE = 1
8     if (x ~ /hello/) {
9         print "second check"
10    }
11 }
12 $ awk -f script.awk
13 second check

```

As visible, the first check fails as the regex is case sensitive, however the second check passes as we have set the `IGNORECASE` variable to a non-zero value.

## 11.8 Control Structures

Just like any programming language, awk supports control structures like if-else, for, while, and do-while loops, which can be used to control the flow of the program and create programs for any use case.

### 11.8.1 if-else

Conditional branching is done using the `if-else` construct in awk. The syntax is similar to that of other programming languages.

`if`

```
1 $ cat script.awk
2 BEGIN{
3     x = 5
4     if(x > 0){
5         print "Positive"
6     }
7 }
8 $ awk -f script.awk
9 Positive
```

The `if` statement checks if the value of `x` is greater than zero and prints `Positive` if it is. Any boolean expression can be used in the `if` statement.

**Remark 11.8.1** A non-zero value is considered true and a zero value is considered false in awk, similar to other programming languages and unlike shell scripting.

We can also use the regex matching operator in the `if` statement.

```
1 $ cat script.awk
2 BEGIN{
3     x = "Hello"
4     if(x ~ /l+/) {
5         print "Matched"
6     }
7 }
8 $ awk -f script.awk
9 Matched
```

Here we are checking if the string `x` contains one or more `\` characters and printing `Matched` if it does.

### if-else

```

1 $ cat script.awk
2 {
3     if($1 > 0){
4         print "Positive"
5     } else {
6         print "Negative"
7     }
8 }
9 $ awk -f script.awk <<< "1"
10 Positive
11 $ awk -f script.awk <<< "-1"
12 Negative

```

Now we can conditionally perform one block of code or another based on the value of the expression. Since boolean expressions can have only two values, we can use the `else` keyword to execute a block of code if the expression evaluates to false. This ensures that one of the two blocks of code is always executed.\*

However, note that not all real world scenarios are boolean in nature. For example, all numbers cannot be classified as either positive or negative. Since zero is neither positive nor negative. We can use the `else if` construct to handle such cases.

### if-else if

```

1 $ cat script.awk
2 {
3     if($1 > 0){

```

---

\* Unless there is a runtime error or the program is terminated.

```

4     print "Positive"
5 } else if($1 < 0){
6     print "Negative"
7 } else {
8     print "Zero"
9 }
10}
11$ awk -f script.awk <<< "1"
Positive
12$ awk -f script.awk <<< "-1"
Negative
14$ awk -f script.awk <<< "0"
15Zero
16

```

Now the program can handle all cases, positive, negative, and zero.

We can add as many `else if` blocks as required to handle all the cases.

The program will try to find the first block whose condition is true and execute that block. If no block is found then the `else` block is executed.

If a block is found then the program will not check the conditions of the other blocks.

```

1 $ cat script.awk
2 BEGIN {
3     x = 15
4     if(x % 5 == 0){
5         print "Divisible by 5"
6     } else if(x % 3 == 0){
7         print "Divisible by 3"
8     }
9     else if(x % 2 == 0){
10        print "Divisible by 2"
11    }
12}
13$ awk -f script.awk
14Divisible by 5

```

Even though the number is divisible by 3 and 5, only the first block is executed as the program stops checking the conditions once a block is matched.

### 11.8.2 Ternary Operator

The ternary operator is a shorthand for the `if-else` construct. It is used to assign a value to a variable based on a condition.

```

1 $ cat script.awk
2 {
3     print ($1 > 0) ? "Positive" : "Negative"
4 }
5 $ awk -f script.awk <<< "1"
6 Positive
7 $ awk -f script.awk <<< "-1"
8 Negative

```

### 11.8.3 for loop

The `for` loop is used to iterate over a sequence of values.

AWK includes both the C-style `for` loop and the `for-in` loop.

#### C-style for loop

```

1 $ cat script.awk
2 BEGIN{
3     for(i = 1; i <= 5; i++){
4         print i
5     }
6 }
7 $ awk -f script.awk
8 1
9 2

```

```

10| 3
11| 4
12| 5

```

The `for` loop has three parts separated by semicolons. The first part is the initialization, the second part is the condition, and the third part is the increment or decrement.

When the loop is first encountered, the initialization part is executed. Then the condition is checked. If the condition is true, then the block of code is executed. After the block of code is executed, the increment part is executed. Then the condition is checked again. This process is repeated until the condition is false.

This means that the variable `i` is incremented by one in each iteration and the loop runs until `i` is less than or equal to 5. After the loop is done, the value of `i` is 6.<sup>5</sup>

However, the value of `i` can be changed inside the loop as well.

5: If this is not clear why the value of `i` is 6, try dry-running the code on a piece of paper to get hang of how C-style for loops work.

```

1 $ cat script.awk
2 BEGIN{
3     for(i = 1; i <= 5; i++){
4         print i
5         if(i == 3){
6             i = 6
7         }
8     }
9 }
10 $ awk -f script.awk
11 1
12 2
13 3

```

The update rule can have any kind of operation, not just increments.

```

1 $ cat script.awk
2 BEGIN{
3     for(i = 1; i <= 1024; i *= 2){
4         print i
5     }
6 }
7 $ awk -f script.awk
8
9 1
10 2
11 4
12 8
13 16
14 32
15 64
16 128
17 256
18 512
19 1024

```

Here we are doubling the value of `i` in each iteration to print the first ten powers of 2.

### for-in loop

The `for-in` loop is used to iterate over the elements of an array. However, awk does not have indexed arrays, so the `for-in` loop is used to iterate over the keys of an associative array.

**Remark 11.8.2** The order of the keys in an associative array is not guaranteed, so the order of the keys in the `for-in` loop is not guaranteed.

```

1 $ cat script.awk
2 BEGIN {
3     assoc["key1"] = "val1"
4     assoc["key2"] = "val2"
5     assoc["key3"] = "val3"
6     for (key in assoc)

```

```

7      print key, assoc[key];
8 }
9 $ awk -f script.awk
10 key3 val3
11 key1 val1
12 key2 val2

```

Here we are iterating over the keys of the associative array `assoc` and printing the key and value of each element. The order of the keys is not guaranteed to be same as the order of insertion.

#### 11.8.4 Iterating over fields

The biggest use-case of for-loops in awk is to iterate over the fields of a record.

```

1 $ cat script.awk
2 {
3     for(i = 1; i <= NF; i++){
4         print i, $i
5     }
6 }
7 $ grep '^[^#]' /etc/passwd | head -n1
8 nobody:*:-2:-2:Unprivileged User:/var/empty:/
    usr/bin/false
9 $ grep '^[^#]' /etc/passwd | head -n1 | awk -F
    : -f script.awk
10 1 nobody
11 2 *
12 3 -2
13 4 -2
14 5 Unprivileged User
15 6 /var/empty
16 7 /usr/bin/false

```

### 11.8.5 while loop

The while loop in awk is similar to the while loop in other programming languages.

```

1 $ cat script.awk
2 BEGIN{
3     i = 1
4     while(i <= 5){
5         print i
6         i++
7     }
8 }
9 $ awk -f script.awk
10 1
11 2
12 3
13 4
14 5

```

The while loop is used to execute a block of code as long as the condition is true. This is useful if you do not know how many times the loop is supposed to run. We can also use the `break` and `continue` statements inside the loop to break out of the loop or skip the current iteration.

```

1 $ cat script.awk
2 BEGIN{
3     i = 1
4     while(i <= 10){
5         if(i == 3){
6             i++
7             continue
8         }
9         print i
10        i++
11        if(i == 5){
12            break
13        }
14    }

```

```
15 }
16 $ awk -f script.awk
17 1
18 2
19 4
```

This script skips the iteration when `i` is 3 and breaks out of the loop when `i` is 5.

We can also match regex inside the condition.

```
1 $ cat script.awk
2 {
3     i = 1
4     while($i ~ /^[A-Z]/){
5         print i, $i
6         i++
7     }
8 }
9 $ cat data.txt
10 Hello World
11 This is a test
12 $ awk -f script.awk data.txt
13 1 Hello
14 2 World
15 1 This
```

The script prints the fields of each record as long as the field starts with an uppercase letter.

### 11.8.6 do-while loop

The **do while** loop is a variation of the `while` looping statement. The `do` loop executes the body once and then repeats the body as long as the condition is true.

```
1 $ cat script.awk
2 BEGIN{
3     i = 1
4     do {
```

```

5   print i
6   i++
7 } while(i <= 5)
8 }
9 $ awk -f script.awk
10 1
11 2
12 3
13 4
14 5

```

This means that even if the condition is false, the body is executed at least once. There is no difference with the while loop after the first iteration.

```

1 $ cat script.awk
2 BEGIN{
3   i = 6
4   do {
5     print i
6     i++
7   } while(i <= 5)
8 }
9 $ awk -f script.awk
10 6

```

### 11.8.7 switch-case

**Switch Case** is a control structure that allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each case.

**Remark 11.8.3** Switch case is a **gawk** extension and is not available in POSIX awk.

The switch statement allows the evaluation of an expression and the execution of statements based on a case match. Case statements are checked for a

match in the order they are defined. If no suitable case is found, the default section is executed, if supplied.

Each case contains a single constant, be it numeric, string, or regexp. The switch expression is evaluated, and then each case's constant is compared against the result in turn.

### Syntax:

```
1 switch (expression) {
2     case value or regular expression:
3         case-body
4     default:
5         default-body
6 }
```

Control flow in the switch statement works as it does in C. Once a match to a given case is made, the case statement bodies execute until a `break`, `continue`, `next`, `nextfile`, or `exit` is encountered, or the end of the switch statement itself.

We will discuss `next` and `nextfile` in a later subsection.

```
1 $ cat script.awk
2 {
3     switch($0){
4         case 1:
5             print "One"
6             break
7         case 2:
8             print "Two"
9             break
10        case 3:
11            print "Three"
12            break
13        case 4:
14            print "Four"
15            break
16    }
17 }
```

```

16    case 5:
17        print "Five"
18        break
19    default:
20        print "Default"
21    }
22}
23$ gawk -f script.awk <<< 5
24Five
25$ gawk -f script.awk <<< 2
26Two
27$ gawk -f script.awk <<< 6
28Default

```

Here we are matching the value of the record against the cases and printing the corresponding value. Observe that we need to specify the keyword `break` after each case. If we however do not use it, then the program will continue to execute the next case as well.

```

1 $ cat script.awk
2 {
3     switch($0){
4         case 1:
5             print "One"
6         case 2:
7             print "Two"
8         case 3:
9             print "Three"
10        case 4:
11            print "Four"
12        case 5:
13            print "Five"
14        default:
15            print "Default"
16    }
17}
18$ gawk -f script.awk <<< 5
19Five
20Default

```

```

21 $ gawk -f script.awk <<< 2
22 Two
23 Three
24 Four
25 Five
26 Default

```

This behaviour, although might seem counter-intuitive, is useful in some cases. For example, if we want to execute the same block of code for multiple cases, then we can do so without repeating the code.

```

1 $ cat script.awk
2 {
3     switch($0 % 10){
4         case 0:
5             print "Multiple of 10"
6         case 5:
7             print "Multiple of 5"
8             break
9         default:
10            print "Not a multiple of 5 or 10"
11    }
12 }
13 $ gawk -f script.awk <<< 15
14 Multiple of 5
15 $ gawk -f script.awk <<< 20
16 Multiple of 10
17 Multiple of 5
18 $ gawk -f script.awk <<< 7
19 Not a multiple of 5 or 10

```

However, unlike the `break` keyword, the `continue` keyword does not directly affect the `switch` statement, but rather the loop that encloses it.

```

1 $ cat script.awk
2 BEGIN {
3     for (i = 1; i <= 3; i++) {
4         switch (i) {
5             case 1:
6                 print "Case 1"

```

```

7         continue
8     case 2:
9         print "Case 2"
10        break
11    case 3:
12        print "Case 3"
13    }
14    print "End of loop iteration", i
15}
16}
17$ gawk -f script.awk
18Case 1
19Case 2
20End of loop iteration 2
21Case 3
22End of loop iteration 3

```

## Matching Regex

We can also match regex in the case statement. This is unlike most programming languages where only constants are allowed.

```

1 $ cat script.awk
2 {
3     switch($0){
4         case /^[A-Z]+$/:
5             print "Strictly Uppercase"
6             break
7         case /^[a-z]+$/:
8             print "Strictly Lowercase"
9             break
10        default:
11            print "Neither"
12        }
13    }
14 $ gawk -f script.awk <<< "HELLO"
15 Strictly Uppercase
16 $ gawk -f script.awk <<< "hello"
17 Strictly Lowercase

```

```
18 $ gawk -f script.awk <<< "Hello"
19 Neither
```

### 11.8.8 break

The break statement jumps out of the innermost for, while, or do loop that encloses it.

```
1 $ cat script.awk
2 {
3     num = $1
4     for (divisor = 2; divisor * divisor <= num
5         ; divisor++) {
6         if (num % divisor == 0)
7             break
8     }
9     if (num % divisor == 0)
10        printf "Smallest divisor of %d is %d\n"
11        ", num, divisor
12    else
13        printf "%d is prime\n", num
14    }
15 $ awk -f script.awk <<< 15
16 Smallest divisor of 15 is 3
15 $ awk -f script.awk <<< 17
16 17 is prime
```

Here we are finding the smallest divisor of a number. If the number is prime, then the smallest divisor is the number itself.

### 11.8.9 continue

The continue statement skips the rest of the loop body and starts the next iteration of the loop.

```
1 $ cat script.awk
2 {
3     for (i = 1; i <= 10; i++) {
```

```

4      if (i % 2 == 0)
5          continue
6      print i
7  }
8 }
9 $ awk -f script.awk
10 1
11 3
12 5
13 7
14 9

```

Here we are printing all the odd numbers from 1 to 10.

### 11.8.10 next and nextfile

**Definition 11.8.1** (next) The `next` statement forces awk to immediately stop processing the current record and go on to the next record. This means that no further rules are executed for the current record, and the rest of the current rule's action isn't executed.

This is similar to how `continue` skips the iteration in a for loop. `awk` internally has a loop that reads the input line by line and processes it. The `next` statement skips the current line and moves to the next line.

```

1 $ cat script.awk
2 FNR == NR {
3     a[$0]=1
4     next
5 }
6 $0 in a{
7     print
8 }
9 $ cat fruits.txt

```

```
10 apple
11 banana
12 orange
13 kiwi
14 coconut
15 $ cat colors.txt
16 red
17 green
18 blue
19 orange
20 yellow
21 purple
22 $ awk -f script.awk fruits.txt colors.txt
23 orange
```

In the above script we use an associative array to store the lines of the first file. Then we check if the line of the second file is present in the associative array. If we do not use the next statement, then the program will print all the lines of the first file as all of them are present in the dictionary. However we are only interested in the lines that are present in the second file as well, so we skip the next check for the lines of the first file.

If the next statement causes the end of the input to be reached, then the code in any END rules is executed. The next statement is not allowed inside *BEGINFILE* and *ENDFILE* rules, we will discuss these rules in a later section.

**Remark 11.8.4** According to the POSIX standard, the behavior is undefined if the next statement is used in a BEGIN or END rule. gawk treats it as a syntax error.

## nextfile

**Definition 11.8.2** (nextfile) The nextfile statement forces awk to immediately stop processing the current input file and go on to the next input file. This means that no further rules are executed for the current line, and the rest of the current rule's action isn't executed. It skips all the remaining lines of the current file and moves to the next file.

Upon execution of the nextfile statement, FILENAME is updated to the name of the next data file listed on the command line, FNR is reset to one, and processing starts over with the first rule in the program. If the nextfile statement causes the end of the input to be reached, then the code in any END rules is executed. An exception to this is when nextfile is invoked during execution of any statement in an END rule; in this case, it causes the program to stop immediately.

```
1 $ cat script.awk
2 BEGINFILE{
3     if(ARGIND==1){
4         pattern = ARGV[1]
5         nextfile
6     }
7 }
8 $0 ~ pattern {
9     print FILENAME, FNR, $0
10 }
11 $ cat colors.txt
12 red
13 green
14 blue
15 orange
16 yellow
17 purple
18 $ cat fruits.txt
19 apple
```

```
20 banana
21 orange
22 kiwi
23 coconut
24 $ gawk -f script.awk orange colors.txt fruits.
25      txt
26 colors.txt 4 orange
fruits.txt 3 orange
```

Note that we need to use `gawk` to run the script as `awk` does not support the `BEGINFILE` statement.

Here we are emulating the `grep` command to search for a pattern in multiple files. We are passing the pattern as the first argument and the files as the rest of the arguments. The `BEGINFILE` rule is executed before the first record of each file. It sets the pattern to the first argument and skips the rest of the file. The `nextfile` statement skips the rest of the file and moves to the next file.

### 11.8.11 exit

The `exit` statement causes `awk` to immediately stop executing the current rule and to stop processing input; any remaining input is ignored. It can also optionally take an exit status, which is returned to the calling process.

**Remark 11.8.5** When an `exit` statement is executed from a `BEGIN` rule, the program stops processing everything immediately. No input records are read. However, if an `END` rule is present, as part of executing the `exit` statement, the `END` rule is executed.

### 11.8.12 BEGINFILE and ENDFILE

Two special kinds of rule, `BEGINFILE` and `ENDFILE`, give you "hooks" into gawk's command-line file processing loop. As with the `BEGIN` and `END` rules, `BEGINFILE` rules in a program execute in the order they are read by gawk. Similarly, all `ENDFILE` rules also execute in the order they are read.

The bodies of the `BEGINFILE` rules execute just before gawk reads the first record from a file. `FILENAME` is set to the name of the current file, and `FNR` is set to zero.

We have already seen an example of the `BEGINFILE` rule in the previous section. `BEGINFILE` are very useful to check if a given argument is a file or not. It can also be used to perform error checking on unreadable files.

The `ENDFILE` rule is called when gawk has finished processing the last record in an input file. For the last input file, it will be called before any `END` rules. The `ENDFILE` rule is executed even for empty input files.

**Remark 11.8.6** The `BEGINFILE` and `ENDFILE` rules are a gawk extension and are not available in POSIX awk.

The `next` statement is not allowed inside either a `BEGINFILE` or an `ENDFILE` rule. The `nextfile` statement is allowed only inside a `BEGINFILE` rule, not inside an `ENDFILE` rule.

## 11.9 Printing

In awk, majority of the output is done using the `print` and `printf` functions.

### 11.9.1 print

The `print` statement is used to print the output to the standard output. It can take multiple arguments and prints them separated by the value of the `OFS` variable. The `print` statement automatically appends a newline character at the end of the output.

```

1 $ cat script.awk
2 BEGIN{
3     print "Hello", "World"
4 }
5 $ awk -f script.awk
6 Hello World
7 $ awk -v OFS="," -f script.awk
8 Hello,World

```

Note that the `OFS` variable is set to a space by default.

The quotes are required, otherwise the `print` statement will treat them as variables, which being undefined will be replaced by an empty string.

In awk if two strings are separated by a space, then they are concatenated.

```

1 $ cat script.awk
2 BEGIN{
3     print "Hello" "World"
4 }
5 $ awk -f script.awk
6 HelloWorld

```

There is no special syntax for concatenation, just place the strings next to each other. This is useful when we have a lot of variables to interpolate with strings.

```

1 $ cat script.awk
2 BEGIN{
3     OFS="\t"
4 }
5 {
6     sum1 += $1
7     sum2 += $2
8     sum3 += $3
9 }
10 END{
11     print "", "Column1", "Column2", "Column3"
12     print "Sum", sum1, sum2, sum3
13     print "Mean", sum1/NR, sum2/NR, sum3/NR
14 }
15 $ cat data.csv
16 12 21 42
17 11 22 45
18 14 27 40
19 $ awk -f script.awk data.csv
20         Column1 Column2 Column3
21 Sum      37      70      127
22 Mean    12.3333 23.3333 42.3333

```

### 11.9.2 printf

awk also supports **formatted printing** using the `printf` function. The `printf` function is similar to the `printf` function in C. It takes a format string and a list of arguments to print.

```

1 $ cat script.awk
2 BEGIN{
3     printf "Hello %s\n", "World"
4 }
5 $ awk -f script.awk

```

## 6 | Hello World

The format string can contain conversion specifiers that start with a percent sign. The conversion specifier is replaced by the corresponding argument. The conversion specifier can also contain flags, width, and precision.

```
1 $ cat script.awk
2 BEGIN{
3     printf "Hello %10s\n", "World"
4 }
5 $ awk -f script.awk
6 Hello      World
```

Here the `%10s` specifier prints the string in a field of width 10. If the string is less than 10 characters, then the remaining characters are filled with spaces.

Similarly, we can format decimal number as well.

```
1 $ cat script.awk
2 BEGIN{
3     printf "The value of pi is %.2f\n", 3.14159
4 }
5 $ awk -f script.awk
6 The value of pi is 3.14
```

The format specifiers are listed in Table 11.1.

**Table 11.1:** Awk Format Specifiers

Format Specifier	Description
a, A	Hexadecimal floating-point number
c	ASCII character
d, i	Decimal integer
e, E	Floating-point number in scientific notation
f	Floating-point number
g, G	Use %e or %f, whichever is shorter
o	Unsigned octal value
s	String
u	Unsigned decimal integer
x, X	Unsigned hexadecimal integer
%	Literal %

Similarly, we can use modifiers to format the output. The modifiers appear after the % and before the conversion specifier.

- ▶ **Flags:** The flags modify the output format. The flags are:

- -: Left-justify the output.
- +: Always print a sign character.
- 0: Pad the output with zeros.
- : If the number is positive, print a space character.
- #: Use an alternative form" for certain control letters.
  - \* For %o, supply a leading zero.
  - \* For %x and %X, supply a leading 0x or 0X for a nonzero result.
  - \* For %e, %E, %f, and %F, the result always contains a decimal point.
  - \* For %g and %G, trailing zeros are not removed from the result.
- ',: For decimal conversions, insert commas every three digits to the left of the decimal point (thousands separator).

- ▶ **Width:** The width specifies the minimum number of characters to print.
- ▶ **Precision:** A period followed by an integer constant specifies the precision to use when printing. The precision specifies the number of decimal places to print. The meaning varies depending on the conversion specifier.
  - %d, %i, %o, %u, %x, %X: Minimum number of digits to print.
  - %e, %E, %f, %F: Number of digits to the right of the decimal point.
  - %g, %G: Maximum number of significant digits.
  - %s: Maximum number of characters from the string that should print.

### 11.9.3 OFS

As we saw earlier, the `print` statement prints the output separated by the value of the `OFS` variable. The `OFS` variable is the output field separator and is set to a space by default. We can set the `OFS` variable to any value we want.

```

1 $ cat script.awk
2 BEGIN{
3     OFS = ","
4     print "Hello", "World"
5 }
6 $ awk -f script.awk
7 Hello,World

```

It can also be changed using the `-v` option.

```

1 $ awk -v OFS="," 'BEGIN{print "Hello", "World
    "}'
2 Hello,World
3 $ awk -v OFS="\t" 'BEGIN{print "Hello", "World
    "}'

```

```
4 | Hello    World
```

We can also change the value while iterating over the records.

```
1 | $ cat script.awk
2 | NR%2==0{
3 |   OFS = ","
4 |   print $1, $2, $3
5 |
6 | NR%2==1{
7 |   OFS = "\t"
8 |   print $1, $2, $3
9 |
10 | $ cat data.ssv
11 | 12 21 42
12 | 11 22 45
13 | 14 27 40
14 | $ awk -f script.awk data.ssv
15 | 12      21      42
16 | 11,22,45
17 | 14      27      40
```

#### 11.9.4 ORS

The `ORS` variable is the output record separator and is set to a newline by default. It is used to separate the records in the output.

```
1 | $ cat script.awk
2 | BEGIN{
3 |   ORS = ","
4 |
5 |   1
6 | $ cat data.txt
7 | Hello
8 | World
9 | $ awk -f script.awk data.txt
10 | Hello,World,
```

**Remark 11.9.1** The **1** is a shorthand for `{print}`. It is used to print the record as it is.

### Concatenating n records

We can use the `ORS` variable to concatenate the output of multiple records by conditionally changing it.

```
1 $ cat script.awk
2 {
3     ORS = NR % 5 ? "," : "\n"
4     print
5 }
6 $ seq 10 | awk -f script.awk
7 1,2,3,4,5
8 6,7,8,9,10
```

Here we are printing the records separated by a comma and a newline after every 5 records. The `NR % 5` is used to check if the record number is a multiple of 5. If it is not, then we print a comma, otherwise we print a newline.

## 11.10 Files and Command Line Arguments

Awk can take any number of arguments to its command line. All the arguments are assumed to be relative or absolute paths to files to be processed. The files are processed in the order they are passed to the command line, left to right.

```
1 $ cat script.awk
2 {
3     print FILENAME, $0
```

```

4 }
5 $ awk -f script.awk fruits.txt colors.txt
6 fruits.txt apple
7 fruits.txt banana
8 fruits.txt orange
9 fruits.txt kiwi
10 fruits.txt coconut
11 colors.txt red
12 colors.txt green
13 colors.txt blue
14 colors.txt orange
15 colors.txt yellow
16 colors.txt purple

```

However, we can also access the command line arguments using the ARGV array.

```

1 $ cat script.awk
2 BEGIN{
3     print ARGV[1]
4     print ARGV[2]
5 }
6 $ awk -f script.awk fruits.txt colors.txt
7 fruits.txt
8 colors.txt

```

### 11.10.1 FILENAME

The `FILENAME` variable contains the name of the current input file. This variable is automatically updated by `awk` as it reads the input files. As soon as the input file changes, the `FILENAME` variable is updated.

In `gawk`, the `BEGINFILE` rule is executed after the `FILENAME` variable is updated. So we can use the `FILENAME` variable to check if the argument is really a file or not.

### 11.10.2 ARGC and ARGV

The command-line arguments available to awk programs are stored in an array called ARGV. ARGC is the number of command-line arguments present. Unlike most awk arrays, ARGV is indexed from 0 to ARGC – 1.

```

1 $ cat script.awk
2 BEGIN{
3     for(i = 0; i < ARGC; i++){
4         print ARGV[i]
5     }
6 }
7 $ awk -f script.awk fruits.txt colors.txt
8 awk
9 fruits.txt
10 colors.txt

```

The names ARGC and ARGV, as well as the convention of indexing the array from 0 to ARGC – 1, are derived from the C language's method of accessing command-line arguments.

**Remark 11.10.1** The ARGV array does not include the flags passed to awk, nor does it include the path to the script file, or the script text.

## 11.11 Input using getline

The `getline` function is used to read input from a file or a pipe. The `getline` function is used in several different ways, such as follows.

- ▶ Getline without arguments
- ▶ Getline into a variable
- ▶ Getline with input file

- ▶ Getline into a variable from a file
- ▶ Getline with a pipe
- ▶ Getline into a variable from a pipe
- ▶ Getline with a coprocess
- ▶ Getline into a variable from a coprocess

We will briefly cover each of these methods.

### 11.11.1 getline without arguments

The getline function can be used without arguments to read input from the current input file. It reads the next record from the current input file and assigns it to the `$0` variable. It also splits it up into the fields as per the value of the `FS` variable.

This is useful if you've finished processing the current record, but want to do some special processing on the next record right now. This form of the getline function sets `NF`, `NR`, `FNR`, `RT`, and the value of `$0`.

This is not the same as the `next` statement, which skips the current record and starts processing the next record from the first rule.

**Remark 11.11.1** The new value of `$0` is used to test the patterns of any subsequent rules. The original value of `$0` that triggered the rule that executed `getline` is lost. By contrast, the `next` statement reads a new record but immediately begins processing it normally, starting with the first rule in the program.

```
1 | $ cat script.awk
2 | #!/usr/bin/env gawk -f
3 |
4 | /start/{
```

```

5     started = 1
6 }
7 {
8     # Keep reading lines until we find one
9     # without the word "skip"
10    while (started && $0 !~ /stop/) {
11        # Read the next line
12        getline
13    }
14    started = 0
15    # Print the line if it doesn't contain "
16    # skip"
17    print $0
18 }
19 $ cat data.txt
20 This is the first line
21 of the file.
22 However, in the second line
23 we may start to ignore some lines.
24 We are still ignoring,
25 as long as we dont encounter stop.
26 This is last line.
27 $ ./script.awk data.txt
28 This is the first line
29 of the file.
30 However, in the second line
31 as long as we dont encounter stop.
32 This is last line.

```

Observe that the `getline` function reads the next line from the file.

### 11.11.2 `getline` into variable

We can also use the `getline` function to read the next line into a variable.

You can use `getline var` to read the next record from awk's input into the variable `var`. No other processing is done. For example, suppose the next

line is a comment or a special string, and you want to read it without triggering any rules. This form of `getline` allows you to read that line and store it in a variable so that the main read-a-line-and-check-each-rule loop of awk never sees it.

```

1 $ cat script.awk
2 {
3     if ((getline tmp) > 0) {
4         print tmp
5         print $0
6     } else
7         print $0
8 }
9 $ cat data.txt
10 Hello
11 World
12 Swap
13 Lines
14 $ awk -f script.awk data.txt
15 World
16 Hello
17 Lines
18 Swap

```

This script reads the next line and prints it before the current line. This effectively swaps every two consecutive lines.

The `getline` function used in this way sets only the variables `NR`, `FNR`, and `RT` (and, of course, `var`). The record is not split into fields, so the values of the fields (including `$0`) and the value of `NF` do not change.

### 11.11.3 `getline` with input file

We can also read from any arbitrary file using the `getline` function, using **redirection**.

```

1 $ cat script.awk
2 {
3     if($0 ~ /file/){
4         getline < "data2.txt"
5     }
6     print
7 }
8 $ cat data.txt
9 This is the first line
10 of the first
11 file.
12 This is second line.
13 $ cat data2.txt
14 Hello world
15 $ awk -f script.awk data.txt
16 This is the first line
17 of the first
18 Hello world
19 This is second line.

```

Here we are reading the next line from `data2.txt` when the current line contains the word `file`. This does not change the value of `NR` and `FNR`. However, it reads the next line from the file and splits it according to `FS` and updates the value of `$0` and `$NF`.

#### 11.11.4 getline into variable from file

Similar to reading the next line into a variable, we can also read from a file into a variable. In this version, no variables are set by the `getline` function.

```

1 $ cat script.awk
2 1
3 /file/{
4     getline tmp < "data2.txt"
5     print tmp
6 }

```

```

7 $ cat data.txt
8 We can print the next line of another
9 file anytime we want:
10 Lets print the next line
11 of the file.
12 $ cat data2.txt
13 Line1
14 Line2
15 $ awk -f script.awk data.txt
16 We can print the next line of another
17 file anytime we want:
18 Line1
19 Lets print the next line
20 of the file.
21 Line2

```

Here we are reading the next line from `data2.txt` when the current line contains the word `file`. Observe that multiple calls to `getline` will read the next line from the file. This means that `awk` maintains a pointer to the file and reads the next line each time `getline` is called.

This is important when we use `getline` multiple times but may want to read the same line again.

To ensure that the file is read from the beginning each time, we can close the file using the `close` function.

```

1 $ cat script.awk
2 {
3     if (NF == 2 && $1 == "@include") {
4         while ((getline line < $2) > 0)
5             print line
6         close($2)
7     } else
8         print
9 }
10 $ cat data.txt
11 This file is the parent file
12 Here we can include lines from other files

```

```

13 @include lib1.txt
14 and from another file
15 @include lib2.txt
16 Repeataions are allowed
17 @include lib1.txt
18 $ cat lib1.txt
19 Library1:
20 lorem ipsum
21 dolor sit amet
22 $ cat lib2.txt
23 Library2:
24 lorem ipsum dolor
25 sit amet
26 $ awk -f script.awk data.txt
27 This file is the parent file
28 Here we can include lines from other files
29 Library1:
30 lorem ipsum
31 dolor sit amet
32 and from another file
33 Library2:
34 lorem ipsum dolor
35 sit amet
36 Repeataions are allowed
37 Library1:
38 lorem ipsum
39 dolor sit amet

```

### 11.11.5 getline with pipe

Using pipes, we can also execute any arbitrary shell command and read its output using command | getline. In this case, the string command is run as a shell command and its output is piped into awk to be used as input. This form of getline reads one record at a time from the pipe.

```

1 $ cat script.awk
2 {
3     if($0 ~ /@date/){

```

```

4      "date" | getline
5  }
6  print
7 }
8 $ cat data.txt
9 This is the first line
10 where we print the date
11 @date
12 Last line
13 $ awk -f script.awk data.txt
14 This is the first line
15 where we print the date
16 Mon Nov  4 18:17:50 IST 2024
17 Last line

```

Here we are reading the output of the date command into the variable \$0, when the current line contains the word `@date`.

When using pipes, `awk` keeps the pipe open even if we read just a single line. We can read more than one line using the same `getline` command, which resumes reading from the original execution instead of running the command again.

```

1 $ cat script.awk
2 /^@get/{
3     "seq 5" | getline
4 }
5 1
6 $ cat data.txt
7 This is the first line
8 where we get the sequence
9 @get
10 We can continue reading using
11 @get
12 and
13 @get
14 and so on
15 $ awk -f script.awk data.txt
16 This is the first line

```

```
17 where we get the sequence
18 1
19 We can continue reading using
20 2
21 and
22 3
23 and so on
```

Here we are reading the output of the seq 5 command into the variable \$0, when the current line contains the word @get. Awk by default keeps the pipe open and resumes reading from the pipe. However, to force the pipe to close, we can use the close function.

```
1 $ cat script.awk
2 /^@get/{
3     "seq 5" | getline
4     close("seq 5")
5 }
6 1
7 $ cat data.txt
8 This is the first line
9 where we get the sequence
10 @get
11 We can continue reading using
12 @get
13 and
14 @get
15 and so on
16 $ awk -f script.awk data.txt
17 This is the first line
18 where we get the sequence
19 1
20 We can continue reading using
21 1
22 and
23 1
24 and so on
```

In this case, the pipe gets closed after reading the

first line. So in the next call to `getline`, the command is executed again.

Using this, we can create a loop to read the output of a command multiple times.

```

1 $ cat script.awk
2 {
3     if ($1 == "@execute") {
4         tmp = substr($0, 10)           # Remove "
5             @execute"
6         while ((tmp | getline) > 0)
7             print
8         close(tmp)
9     } else
10    print
11 }
12 $ cat data.txt
13 We can see our username using
14 @execute whoami
15 and we can see who all are logged in using
16 @execute who
17 $ awk -f script.awk data.txt
18 We can see our username using
19 sayan
20 and we can see who all are logged in using
21 sayan          console      Oct 28 11:50
22 sayan          ttys000     Oct 30 02:53

```

This variation of `getline` splits the record into fields, sets the value of `NF`, and recomputes the value of `$0`. The values of `NR` and `FNR` are not changed. `RT` is set.

### 11.11.6 `getline` into a variable from a pipe

Similarly, we can read the output of a command into a variable.

```

1 $ cat script.awk
2 {

```

```

3  if($0 ~ /@date/){
4      "date" | getline tmp
5      print tmp
6  }
7  print
8 }
9 $ cat data.txt
10 This is the first line
11 where we print the date
12 @date
13 Last line
14 $ awk -f script.awk data.txt
15 This is the first line
16 where we print the date
17 Mon Nov  4 18:50:55 IST 2024
18 @date
19 Last line

```

In this version, we can retain the original line as `$0` is not overwritten.

This is useful when we want to read the output of a command and use it in the same record.

In this version of `getline`, none of the predefined variables are changed and the record is not split into fields. However, `RT` is set.

### 11.11.7 `getline` with coprocess

Reading input into `getline` from a pipe is a one-way operation. The command that is started with `command | getline` only sends data to your awk program.

On occasion, you might want to send data to another program for processing and then read the results back. `gawk` allows you to start a coprocess, with which two-way communications are possible. This is done with the `|&` operator. Typically, you write

data to the coprocess first and then read the results back, as shown in the following:

```
1 | print "some query" |& "db_server"
2 | "db_server" |& getline
```

which sends a query to db\_server and then reads the results.

This can also be done using a variable.

**Remark 11.11.2** This is a gawk extension and is not available in POSIX awk.

## 11.12 Redirection and Piping

In awk, we can use **redirection** and **piping** to read from files and write to files.

There are four forms of output redirection:

- ▶ output to a file,
- ▶ output appended to a file,
- ▶ output through a pipe to another command, and
- ▶ output to a coprocess.

### 11.12.1 Output Redirection

```
1 | print "Hello, World" > "output.txt"
```

This is different from how you use redirections in shell scripts.

When this type of redirection is used, the output-file is erased before the first output is written to it. Subsequent writes to the same output-file do not erase output-file, but append to it. If output-file does not exist, it is created.

```
1 $ cat script.awk
2 {
3     print $1 > "firstnames.txt"
4     print $2 > "lastnames.txt"
5 }
6 $ cat names.txt
7 John Doe
8 Jane Smith
9 $ awk -f script.awk names.txt
10 $ cat firstnames.txt
11 John
12 Jane
13 $ cat lastnames.txt
14 Doe
15 Smith
```

## 11.12.2 Appending

```
1 | print "Hello, World" >> "output.txt"
```

This redirection prints the items into the preexisting output file named output-file. The difference between this and the single-> redirection is that the old contents (if any) of output-file are not erased. Instead, the awk output is appended to the file. If output-file does not exist, then it is created.

## 11.12.3 Piping to other commands

It is possible to send output to another program through a pipe instead of into a file. This redirection opens a pipe to the command, and writes the output through this pipe to another process created to execute the command.

```
1 $ cat script.awk
2 {
3     print $0 | "wc -w"
```

```

4 }
5 $ cat data.txt
6 This is the first line
7 of the file.
8 $ awk -f script.awk data.txt
9 8

```

Even though `awk` runs for each line one by one and sends the output to the `wc -w` command, the `wc -w` command reads the entire input and prints the word count. This is because the pipe remains open until the `awk` program finishes execution, or the pipe is closed using the `close` function.

To print the number of words of each line, we can close the pipe after each line.

```

1 $ cat script.awk
2 {
3     print $0 | "wc -w"
4     close("wc -w")
5 }
6 $ cat data.txt
7 This is the first line
8 of the file.
9 $ awk -f script.awk data.txt
10    5
11    3

```

#### 11.12.4 Piping to a coprocess

This redirection prints the items to the input of command. The difference between this and the single-| redirection is that the output from command can be read with `getline`. Thus, command is a **coprocess**, which works together with but is subsidiary to the `awk` program.

This feature is a gawk extension, and is not available in POSIX awk.

### 11.12.5 system

We can also run shell commands using the `system` function. The `system` function runs the shell command and returns the exit status of the command.

```
1 $ cat script.awk
2 {
3     system("echo \" $0 \" | wc -w" )
4 }
5 $ cat data.txt
6 This is the first line
7 of the file.
8 $ awk -f script.awk data.txt
9      5
10     3
```

Here we are running the `wc -w` command on each line using the `system` function.

**Remark 11.12.1** Note that we do not need to close the command when using the `system` function.

## 11.13 Examples

Finally, to end this long chapter on `awk`, we will go through some examples to understand how to use `awk` in real life scenarios.

### 11.13.1 Finding the Frequency

Awk can be used to find the frequency of words in a file.

```

1 $ cat script.awk
2 BEGIN{
3     FPAT="[A-Za-z0-9]+"
4     OFS=":"
5 }
6 {
7     for(i = 1; i <= NF; i++){
8         words[$i]++
9     }
10 }
11 END{
12     for(word in words){
13         print words[word], word
14     }
15 }
16
17 $ ls -l alice.txt # too big to print
18 -rw-r--r--@ 1 sayan staff 148574 Nov  4
2023:35 alice.txt
19 $ gawk -f script.awk alice.txt | sort -nr |
20     head
21 1527:the
22 802:and
23 725:to
24 615:a
25 545:I
26 527:it
27 509:she
28 500:of
29 456:said
30 396:Alice

```

Let us understand the script.

- ▶ **FPAT:** The `FPAT` variable is a gawk extension that allows you to define the field using a regular expression. Here we are defining a field as any alphanumeric character of any non-zero length.
- ▶ **OFS:** The `OFS` variable is set to a colon to separate the frequency and the word.

- ▶ **words**: This is an associative array that stores the frequency of each word.
- ▶ **END**: At the end, we iterate over the `words` array and print the frequency and the word.
- ▶ **sort -nr**: We sort the output in reverse numerical order.
- ▶ **head**: We print the first 10 lines.

Similarly, we can simplify the script to find frequencies of the lines instead.

```

1 $ cat script.awk
2 {
3     lines[$0]++
4 }
5 END{
6     for(line in lines){
7         print lines[line], line
8     }
9 }
10 $ cat data.txt
11 apple
12 orange
13 apple
14 banana
15 $ awk -f script.awk data.txt
16 2 apple
17 1 orange
18 1 banana

```

## 11.13.2 Descriptive Statistics

We can also use `awk` to calculate the mean, median, and mode of a set of numbers.

```

1 $ cat script.awk
2 {
3     sum += $1
4     numbers[$1]++;
5     lines[NR] = $1

```

```

6 }
7 END{
8   n = asorti(numbers, sorted)
9   mean = sum / NR
10  asort(lines)
11  median = (NR % 2) ? lines[(NR + 1) / 2] : (
12    lines[NR / 2] + lines[NR / 2 + 1]) / 2
13  mode = 0
14  for(i = 1; i <= n; i++){
15    if(numbers[sorted[i]] > mode){
16      mode = sorted[i]
17    }
18  print "Mean:", mean
19  print "Median:", median
20  print "Mode:", mode
21 }
22 $ for i in {1..10000}; do echo $RANDOM ; done
23 > numbers.txt
24 $ ls -l numbers.txt
25 -rw-r--r--@ 1 sayan  staff  56626 Nov  4 23:45
26   numbers.txt
27 $ awk -f script.awk numbers.txt
28 Mean: 16466.5
29 Median: 16418
30 Mode: 5551

```

Here we are calculating the mean, median, and mode of a set of numbers. Let us understand the script.

- ▶ **sum:** We are calculating the sum of all the numbers.
- ▶ **numbers:** This is an associative array that stores the frequency of each number.
- ▶ **lines:** This is an array that stores the numbers in the order they appear.
- ▶ **asort:** This function sorts the array `lines`.
- ▶ **asorti:** This function sorts the array `numbers` and returns the number of elements in the

array.

- ▶ **mean:** We calculate the mean by dividing the sum by the number of elements.
- ▶ **median:** We calculate the median by checking if the number of elements is odd or even.
- ▶ **mode:** We calculate the mode by iterating over the sorted array and finding the number with the highest frequency.

### 11.13.3 Semi-Quoted CSV Handing

We can also handle semi-quoted CSV files using **gawk**'s FPAT variable.

```

1 $ cat script.awk
2 BEGIN{
3     FPAT = "([^\,]+)|(\\"[^\\\"]+\\")"
4     OFS = ","
5 }
6 {
7     for(i = 1; i <= NF; i++){
8         gsub("^[^\,]*$", "\\"&"\", $i)
9     }
10    print
11 }
12 $ cat data.csv
13 a,b,c
14 1,2,3
15 "a word",sometimes quoted,"sometimes, with
   commas"
16 $ gawk -f script.awk data.csv
17 "a","b","c"
18 "1","2","3"
19 "a word","sometimes quoted","sometimes, with
   commas"
```

Here we are handling semi-quoted CSV files and converting them into fully quoted CSV files.

Students are encouraged to go through some other examples listed [here](#) and try to understand how they work. Further it is beneficial to try to create scripts of their own to solve problems they face using **bash**, **sed**, and **awk**.