Of course. Here are comprehensive solutions and alternative approaches for all 55 exercises, designed to explain the underlying concepts crucial for a DevOps professional.

Easy: Foundational Command Fluency

1. Navigation & Inspection * Task: List all files, including hidden, in long format sorted by modification time (newest first). * Solution: `ls -laht` * `-l`: Long listing * `-a`: All (including hidden, which start with `.`) * `-h`: Human-readable file sizes (e.g., 4K, 1.2M) * `-t`: Sort by modification time * Alternative: `ls -larth` (Sorts oldest first, `-r` reverses the order).

2. File Creation * Task: Create `project/{dev,prod,test}/config` with a single command. * Solution: `mkdir -p project/{dev,prod,test}/config` * `-p`: Creates parent directories as needed (no error if existing). * `{a,b,c}`: Brace expansion, a core shell feature for generating strings.

3. Basic File Ops * Task: Copy all `.log` files from `/var/log` to `~/backups/logs`, preserving attributes. * Solution: `cp -a /var/log/*.log ~/backups/logs/` * `-a` (archive mode): Preserves all attributes (ownership, permissions, timestamps). Equivalent to `-dR --preserve=all`. * Alternative: `rsync -a /var/log/*.log ~/backups/logs/` (Better for large files/network transfers, only copies changes).

4. User & Permissions * Task: Change owner of `deploy.sh` to user `deploy` and group `www-data`. Give owner execute and group read. * Solution: `bash sudo chown deploy:www-data deploy.sh chmod u+x,g+r deploy.sh * chown user:group file` * `chmod u/g/o +/- r/w/x file` (user, group, others / add, remove / read, write, execute) * Alternative (Numeric): `chmod 740 deploy.sh` * `7` (user: 4(r)+2(w)+1(x)) * `4` (group: 4(r)) * `0` (others: 0)

5. Text Manipulation (head/tail) * Task: Display first 5 and last 10 lines of `application.log`. * Solution: `(head -5 application.log; tail -10 application.log)` * The parentheses run the commands in a subshell, grouping their output. * Alternative: `tail -10 application.log | cat <(head -5 application.log) -` * Uses process substitution (`<()`) to feed the output of `head` into `cat`. The `-` tells `cat` to read from stdin (the output of `tail`).

6. Searching (grep) * Task: Find lines with `"500"` in `nginx.access.log` and save to `errors.log`. * Solution: `grep "500" nginx.access.log > errors.log` * Alternative (Case-insensitive, count): * `grep -i "internal server error" nginx.access.log >> errors.log` (`-i` ignores case, `>>` appends) * `grep -c "500" nginx.access.log` (Just counts the occurrences)

7. Process Management * Task: Find PID of `nginx` and send it `SIGHUP`. * Solution: `sudo kill -HUP $(pgrep nginx)` * `pgrep nginx`: Finds the PID(s) of processes named `nginx`. * `$(...)`: Command substitution, the output of `pgrep` is

passed as an argument to `kill`. * Alternative: `sudo pkill -HUP nginx` (`pkill` finds and signals processes by name directly). Or use `systemctl reload nginx` if it's a systemd service.

8. Disk Usage * Task: Find top 5 largest files/directories in `/home`. * Solution: `du -ah /home | sort -rh | head -5` * `du -ah`: Disk usage, all files, human-readable. * `sort -rh`: Reverse sort, human-readable numeric sort (so 10M comes before 9K). * `head -5`: Show first 5 lines. * Alternative: `ncdu /home` (Interactive and much faster TUI tool, not a one-liner but highly practical).

9. Downloading * Task: Use `curl` to download a file and save it with a different name. * Solution: `curl -o my_new_file.zip https://example.com/old_file.zip` * `-o`: Write output to `<file>` instead of stdout. * Alternative: `wget -O my_new_file.zip https://example.com/old_file.zip` (Using `wget`).

10. Basic Piping * Task: List running processes, filter for user, and count. * Solution: `ps aux | grep "$USER" | grep -v grep | wc -l` * `ps aux`: Lists all processes. * `grep "$USER"`: Filters for lines containing your username. * `grep -v grep`: Excludes the `grep` command itself from the results. * `wc -l`: Counts lines. * Alternative (more reliable): `pgrep -u $USER | wc -l` (`pgrep` is designed for this).

Medium: Scripting Fundamentals & System Interaction

11. Simple Script Skeleton * Task: Script that checks if a file (from argument) exists. * Solution (**`check_file.sh`**): `bash #!/bin/bash if [[ -f "$1" ]]; then echo "File exists." else echo "File does not exist." fi` * Alternative (using test command): `if test -f "$1"; then ...`

12. For Loop * Task: Ping hosts from `hosts.txt` and report reachability. * Solution (**`ping_hosts.sh`**): `bash #!/bin/bash for host in $(cat hosts.txt); do if ping -c 1 -W 1 "$host" &> /dev/null; then echo "$host is reachable." else echo "$host is unreachable." fi done` * `-c 1`: Send one packet. * `-W 1`: Wait 1 second for a reply. * `&> /dev/null`: Redirects both stdout and stderr to null (suppress all output). * Alternative (using **`while read`** for robustness): `bash while IFS= read -r host; do # ... ping logic here ... done < hosts.txt`

13. User Input * Task: Ask for user's name and greet them. * Solution (**`greeter.sh`**): `bash #!/bin/bash read -p "What is your name? " name echo "Hello, $name!"` * `read -p`: Prints a prompt before reading input.

14. Argument Parsing * Task: Script with `-d` for directory and `-e` for extension to count files. * Solution (**`count_files.sh`**): `bash #!/bin/bash`

```
while getopts "d:e:" opt; do          case $opt in              d)
target_dir="$OPTARG" ;;              e) extension="$OPTARG" ;;
*) echo "Invalid option"; exit 1 ;;          esac     done     find
"$target_dir" -maxdepth 1 -name "*$extension" -type f | wc -l    *
```
`getopts` is the standard way to parse single-character flags. * `find` is more reliable than `ls` for scripting, especially with special characters.

15. System Info Script * Task: Output a system report. * Solution (**`sysreport.sh`**): `bash     #!/bin/bash     echo "=== System Report ==="     echo "Date: $(date)"     echo "Disk Usage:"     df -h / | awk 'NR==2{print $5}'     echo "Memory Usage:"     free -h | awk '/Mem:/{print $3 "/" $2}'     echo "Top 5 CPU Processes:" ps -eo pid,comm,%cpu --sort=-%cpu | head -n 6` * Alternative: Use `top -bn1 | head -n 12` for a snapshot of top.

16. Backup Script * Task: Create a timestamped backup. * Solution (**`backup.sh`**): `bash   #!/bin/bash   source_dir="$1"   backup_dir="/backups" timestamp=$(date +%Y%m%d_%H%M%S)   backup_name="backup_$(basename "$source_dir")_$timestamp.tar.gz"   tar -czf "$backup_dir/$backup_name" -C "$(dirname "$source_dir")" "$(basename "$source_dir")"   echo "Backup created: $backup_dir/$backup_name"` * `-C $(dirname "$source_dir")`: `tar` changes to this directory before archiving, so the archive doesn't contain full paths like `/home/user/project`, just `project`.

17. Log Analyzer * Task: Count HTTP status codes in a log file. * Solution (**`log_analyzer.sh`**): `bash     #!/bin/bash     log_file="$1"     echo "HTTP Status Code Counts:"     grep -oE '" [0-9]{3} ' "$log_file" | sort | uniq -c | sort -nr` * `grep -oE '" [0-9]{3} '`: `-o` outputs only the matching part, `-E` uses extended regex. * `uniq -c`: Counts occurrences of unique lines (must be sorted first).

18. String Manipulation * Task: Extract filename and directory from a full path. * Solution (**`path_parser.sh`**): `bash     #!/bin/bash     full_path="$1" filename=$(basename "$full_path")     dirname=$(dirname "$full_path") echo "Filename (without path): ${filename%.*}" # Removes extension echo "Directory: $dirname"` * Alternative (Parameter Expansion): `bash full_path="/home/user/docs/report.txt"     filename="${full_path##*/}" # report.txt     dirname="${full_path%/*}"   # /home/user/docs basename="${filename%.*}"   # report`

19. API Interaction * Task: Use `curl` and `jq` to parse a JSON API response. * Solution (**`github_user.sh`**): `bash     #!/bin/bash     username="$1" response=$(curl -s "https://api.github.com/users/$username") login=$(echo "$response" | jq -r '.login')     name=$(echo "$response" | jq -r '.name')     echo "Username: $login"     echo "Name: $name"` * `jq -r`: Raw output (removes quotes from strings).

20. Cron Job * Task: Schedule the backup script to run weekly. * Solution: Add this line to your crontab (`crontab -e`): `0 2 * * 0 /path/to/backup.sh`

`/path/to/source_dir >> /var/log/backup.log 2>&1` * `0 2 * * 0`: At 02:00 on Sunday. * `>> /var/log/backup.log 2>&1`: Appends both stdout and stderr to a log file.

21. Function * Task: Create a `log_message` function. * Solution (**`logger.sh`**): `bash #!/bin/bash LOGFILE="./script.log" log_message() { echo "$(date '+%Y-%m-%d %H:%M:%S') - $1" >> "$LOGFILE" } log_message "Script started" # ... do work ... log_message "Script finished"`

22. Check for Tools * Task: Check if `docker`, `git`, and `jq` are installed. * Solution (**`check_deps.sh`**): `bash #!/bin/bash exit_code=0 for cmd in docker git jq; do if ! command -v "$cmd" &> /dev/null; then echo "Error: $cmd is not installed." >&2 exit_code=1 fi done if [[ $exit_code -ne 0 ]]; then exit $exit_code fi echo "All dependencies are met."` * `command -v` is the most reliable way to check if a command exists.

23. File Modification * Task: Comment out lines containing `DEBUG` in `.conf` files. * Solution (**`comment_debug.sh`**): `bash #!/bin/bash find /path/to/configs -name "*.conf" -exec sed -i '/DEBUG/s/^/#/' {} \;` * `find -exec`: Runs the `sed` command on each file found. * `sed -i`: In-place edit. * `/DEBUG/s/^/#/`: For lines matching `DEBUG`, substitute the start of the line (`^`) with a `#`.

24. Interactive Delete * Task: Find files >30 days old and delete interactively. * Solution (**`clean_old.sh`**): `bash #!/bin/bash find /path/to/dir -type f -mtime +30 -exec rm -i {} \;` * `-mtime +30`: Modified time is greater than 30 days. * `rm -i`: Interactive mode, prompts for each file.

25. Port Check * Task: Check if a remote port is open. * Solution (**`check_port.sh`**): `bash #!/bin/bash host="$1" port="$2" timeout 1 bash -c "cat < /dev/null > /dev/tcp/$host/$port" 2>/dev/null if [[ $? -eq 0 ]]; then echo "Port $port on $host is OPEN." else echo "Port $port on $host is CLOSED." fi` * Uses bash's built-in `/dev/tcp` magic. * `timeout 1`: Ensures the command doesn't hang.

Hard: Advanced Scripting & Robustness

26. Error Handling * Task: Make backup script robust with error handling and trapping. * Solution (**`robust_backup.sh`**): `bash #!/bin/bash set -euo pipefail # Exit on error, undefined var, pipefail source_dir="${1:-}" backup_dir="/backups" cleanup() { echo "Cleaning up on exit..." # Remove incomplete backup if it exists rm -f "$backup_dir/${backup_name:-NOTSET}" } trap cleanup`

```
EXIT ERR INT TERM     # Validation     [[ -z "$source_dir" ]] &&
{ echo "Usage: $0 <source_dir>"; exit 1; }     [[ ! -d "$source_dir"
]] && { echo "Error: $source_dir does not exist."; exit 1; }
[[ ! -w "$backup_dir" ]] && { echo "Error: $backup_dir is not
writable."; exit 1; }     timestamp=$(date +%Y%m%d_%H%M%S)     backup_name="backup_$(basenam
"$source_dir")_$timestamp.tar.gz"     if ! tar -czf "$backup_dir/$backup_name"
-C "$(dirname "$source_dir")" "$(basename "$source_dir")"; then
echo "Error: tar command failed." >&2         exit 1     fi
trap - EXIT ERR INT TERM # Disable the trap on successful completion
echo "Backup successful: $backup_dir/$backup_name"
```

27. Configuration Parsing * Task: Parse a simple key=value config file. * Solution (**load_config.sh**): bash

```
#!/bin/bash     config_file="${1:-config.conf}"
# Safely read the config, ignoring comments and empty lines
while IFS= read -r line; do         # Remove everything after a
'#' and trim leading/trailing space         cleaned_line="${line%%#*}"
cleaned_line="${cleaned_line##*( )}" # Trim leading spaces (needs
`shopt -s extglob`)         cleaned_line="${cleaned_line%%*(
)}"  # Trim trailing spaces         if [[ -n "$cleaned_line" &&
"$cleaned_line" == *=* ]]; then             key="${cleaned_line%%=*}"
value="${cleaned_line#*=}"             # Declare the variable (be
cautious with sourcing!)             declare -- "$key"="$value"
echo "Config: $key=$value"         fi     done < <(grep -v '^#\|^$'
"$config_file") # Filter comments/empty lines first     echo
"Loaded value for DB_HOST: ${DB_HOST:-Not Set}" *
```

Alternative (Safer): Don't use `declare`. Instead, use an associative array: `config["key"]="$value"`.

28. JSON/CSV Processing * Task: Process a CSV and create users. * Solution (**create_users.sh**):

```
bash     #!/bin/bash     # CSV format:
username,uid,group     input_file="users.csv"     while IFS=','
read -r username uid group; do         # Basic input sanitization
if [[ -z "$username" || "$username" =~ [^a-zA-Z0-9_] ]]; then
echo "Skipping invalid username: $username" >&2             continue
fi         echo "Creating user: $username with UID:$uid and
group:$group"         # sudo groupadd -f "$group" # Create group
if it doesn't exist         # sudo useradd -u "$uid" -G "$group"
"$username"     done < <(tail -n +2 "$input_file") # Skips the
header line     echo "User creation complete (commands commented
out for safety)." *
```

Crucial: This example has the dangerous commands (`useradd`, `groupadd`) commented out. Always test with `echo` first!

29. Interactive Menu * Task: Create a text-based menu. * Solution (**menu.sh**):

```
bash     #!/bin/bash     while true; do         echo "1. Check
Disk Usage"         echo "2. Check Memory Usage"         echo "3.
Exit"         read -p "Please select an option [1-3]: " choice
case $choice in             1) df -h ;;             2) free -h ;;
3) echo "Exiting..."; exit 0 ;;             *) echo "Invalid
```

```
option. Please try again." ;;         esac         echo # Print a
newline     done
```

30. Log Monitoring * Task: Tail a log and alert on repeated errors. * Solution (**log_monitor.sh**): `bash   #!/bin/bash   log_file="$1"
error_pattern="ERROR"   threshold=5   interval=60   echo
"Monitoring $log_file for '$error_pattern'..."   tail -f "$log_file"
| while read -r line; do   if echo "$line" | grep -q "$error_pattern";
then   count=$((count + 1))   echo "Error
detected. Count: $count"   if ((count >= threshold
)); then   echo "ALERT: $threshold errors detected
in the last minute!"   # | mail -s "Error Alert"
admin@example.com   count=0 # Reset counter after
alert   fi   # Simple reset mechanism using a
background job   (sleep $interval && count=$((count -
1))) &   fi   done` * This is a simplistic approach. A more robust tool like `swatch` or `logwatch` is better for production.

31. Git Automation * Task: Automate git add, commit, push. * Solution (**git_auto.sh**): `bash   #!/bin/bash   # Check if there are
changes to commit   if [[-z $(git status --porcelain)]]; then
echo "No changes to commit."   exit 0   fi   git add
.   commit_message="Auto-commit: $(date '+%Y-%m-%d %H:%M:%S')"
git commit -m "$commit_message"   # Get current branch name
current_branch=$(git symbolic-ref --short HEAD)   git push
origin "$current_branch"`

32. Docker Container Manager * Task: List, stop, and remove containers by name/ID. * Solution (**docker_manager.sh**): `bash   #!/bin/bash   echo
"Running containers:"   docker ps --format "table {{.ID}}\t{{.Names}}"
read -p "Enter container name or ID to stop (or 'quit'): " container_input
if [["$container_input" == "quit"]]; then exit 0; fi   # Stop
the container   echo "Stopping container $container_input..."
docker stop "$container_input"   # Remove the container   read
-p "Remove container $container_input? (y/N): " confirm_remove
if [["$confirm_remove" == "y"]]; then   docker rm "$container_input"
echo "Container removed."   fi`

33. Password Generator * Task: Generate a random, secure password. * Solution (**gen_passwd.sh**): `bash   #!/bin/bash   length=${1:-16}
# Define character sets   upper="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
lower="abcdefghijklmnopqrstuvwxyz"   numbers="0123456789"
symbols='!@#$%^&*()_+-={}[]|:;<>,.?/'   all_chars="${upper}${lower}${numbers}${symbols}"
password=$(head /dev/urandom | tr -dc "$all_chars" | head -c
"$length")   echo "Generated Password: $password"` * `/dev/urandom` is a cryptographically secure random source. * `tr -dc "$all_chars"`: Deletes (`-d`) all characters not in (`-c`) the `$all_chars` set.

34. Parallel Execution * Task: Ping 50 hosts concurrently. * Solution with **parallel** (**ping_parallel.sh**): bash #!/bin/bash cat hosts.txt | parallel -j 20 'ping -c 1 -W 1 {} &> /dev/null && echo {} is UP || echo {} is DOWN' * -j 20: Run 20 jobs in parallel. * Solution with background processes (**ping_bg.sh**): bash #!/bin/bash while IFS= read -r host; do (ping -c 1 -W 1 "$host" &> /dev/null && echo "$host is UP") & done < hosts.txt wait # Wait for all background processes to finish echo "Ping sweep complete."

35. Self-Documenting Script * Task: Script with `getopts` for help, verbose, file. * Solution (**professional_script.sh**): bash #!/bin/bash usage() { echo "Usage: $0 [-v] [-h] -f <config_file>" >&2 echo " -v: Enable verbose mode." echo " -h: Show this help message." echo " -f: Specify configuration file." } VERBOSE=false CONFIG_FILE="" while getopts "vhf:" opt; do case $opt in v) VERBOSE=true ;; h) usage; exit 0 ;; f) CONFIG_FILE="$OPTARG" ;; *) usage; exit 1 ;; esac done # Check mandatory options if [[-z "$CONFIG_FILE"]]; then echo "Error: -f option is required." >&2 usage exit 1 fi [["$VERBOSE" == "true"]] && echo "Verbose mode enabled. Using config: $CONFIG_FILE" # ... main script logic ...

Tricky: Edge Cases & Deep Understanding

36. Safe **rm** * Task: Create a safe `rm` wrapper. * Solution (Add to **.bashrc** or **.bash_aliases**): bash safe_rm() { local trash_dir="${HOME}/.trash" mkdir -p "$trash_dir" for item in "$@"; do # Move to trash with a timestamp to avoid overwrites mv -- "$item" "$trash_dir/$(basename "$item")_$(date +%s)" done echo "Moved $# item(s) to $trash_dir" } # alias rm="safe_rm" # Uncomment with extreme caution! Breaks scripts expecting real rm.

37. String Replacement in Bulk * Task: Replace `old.example.com` with `new.example.com` in `.php` and `.html` files. * Solution: `find /project -type f \( -name "*.php" -o -name "*.html" \) -exec sed -i 's/old\.example\.com/new.example.com/g' {} \;` * `\( -name "*.php" -o -name "*.html" \)`: Logic OR for multiple file extensions. * `s/old\.example\.com/new.example.com/g`: Global substitution. Note the escaped dots `\.`.

38. Validate IP Address * Task: Function to validate an IPv4 address with regex. * Solution: bash validate_ip() { local ip=$1 local stat=1 if [[$ip =~ ^[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}$]]; then # Check each octet is <= 255 IFS='.'

```
read -r -a octets <<< "$ip"             for octet in "${octets[@]}";
do                  if (( octet > 255 )); then                      return
1                 fi              done              stat=0         fi
return $stat     }     # Usage     if validate_ip "192.168.1.1";
then echo "Valid IP"; else echo "Invalid IP"; fi
```

39. Script Locking * Task: Ensure only one instance of a script runs. * Solution (**singleton.sh**): `bash #!/bin/bash LOCKFILE="/tmp/$(basename "$0").lock" if [[-e "$LOCKFILE"]]; then pid=$(cat "$LOCKFILE") if kill -0 "$pid" 2>/dev/null; then echo "Script is already running (PID: $pid). Exiting." >&2 exit 1 else echo "Removing stale lock file." >&2 rm -f "$LOCKFILE" fi fi echo $$ > "$LOCKFILE" trap 'rm -f "$LOCKFILE"; exit' EXIT ERR INT TERM # ... main script body ...` * `kill -0 $pid`: Checks if a process with that PID exists.

40. SSH Automation * Task: Run `df -h` on multiple remote servers. * Solution (**ssh_multi_run.sh**): `bash #!/bin/bash # servers.txt should list one host per line while IFS= read -r server; do echo "=== Disk Usage on $server ===" # Use `-o ConnectTimeout=5` to avoid hanging ssh -o ConnectTimeout=5 -o BatchMode=yes "$server" "df -h /" 2>/dev/null || echo "Failed to connect to $server" echo done < servers.txt` * `-o BatchMode=yes`: Avoids password prompts (requires SSH keys to be set up).

41. The Space Problem * Task: Loop over files correctly, handling special characters. * Solution: `bash find . -maxdepth 1 -type f -print0 | while IFS= read -r -d '' file; do echo "Processing: $file" done` * `-print0` and `read -d ''` use the null character as a delimiter, the only safe way to handle any filename.

42. Quoting Hell * Task: Demonstrate `$@` vs `$*`. * Solution: `bash # Script: test_args.sh echo "Number of args: $#" echo "Using \$@ (with quotes):" for arg in "$@"; do echo "[$arg]"; done echo "Using \$* (with quotes):" for arg in "$*"; do echo "[$arg]"; done # Only one loop iteration! # Run it: ./test_args.sh "arg one" "arg two" # "$@" expands to "arg one" "arg two" (2 elements) # "$*" expands to "arg one arg two" (1 element)`

43. Source vs Execute * Task: Demonstrate sourcing vs executing. * Solution: `script.sh: #!/bin/bash; my_var="Hello from script"` * Execute (**`./script.sh`**): Runs in a subshell. `my_var` is not available in your current shell after it finishes. * Source (**`source script.sh`** or **`. script.sh`**): Runs in the current shell. `my_var` is now defined in your current shell.

44. Subshell Gotcha * Task: Show variable scope issue in piped `while` loop. * Solution: `bash count=0 echo -e "line1\nline2" | while read -r line; do ((count++)) done echo "Count from`

```
subshell: $count" # Prints 0! The pipe creates a subshell.     #
Fix: Use process substitution or a temporary file     count=0
while read -r line; do         ((count++))     done < <(echo
-e "line1\nline2")     echo "Count without subshell: $count" #
Prints 2
```

45. Exit Code Chaining * Task: Chain commands based on exit codes. * Solution:

```
bash     command1 && command2  # Run command2 only if command1
succeeds (exit 0)     command1 || command2  # Run command2 only
if command1 fails (exit non-zero)
```

46. The Null Command * Task: Demonstrate the colon : command. * Solution:

```
bash     : # Does nothing, successfully (exit 0)     # Use
case 1: Placeholder     if some_condition; then         : # TODO:
implement this block later     else         do_something     fi
# Use case 2: No-op for mandatory syntax     while : # infinite
loop     do         sleep 1     done
```

47. Parameter Expansion Magic * Task: Demonstrate various parameter expansions. * Solution:

```
bash     var="hello.world.txt"     echo "Length:
${#var}"           # 14     echo "Remove suffix: ${var%.*}"  #
hello.world     echo "Remove prefix: ${var##*.}" # txt     unset
maybe_empty     echo "Default if unset: ${maybe_empty:-default_value}"
# default_value
```

48. Arithmetic in Bash * Task: Calculate average using only bash. * Solution:

```
bash     # numbers.txt has one number per line     sum=0
count=0     while IFS= read -r num; do         sum=$((sum + num))
count=$((count + 1))     done < numbers.txt     # Bash only does
integer math, so we calculate integer average     average=$((sum
/ count))     echo "Average: $average"     # For floating point,
you MUST use bc or awk: echo "$sum / $count" | bc -l
```

49. Test Builtin * Task: Use [[ ]] for regex and file tests. * Solution:

```
bash     filename="test.txt"     string="Hello123"     if [[
-r "$filename" && -w "$filename" ]]; then         echo "File is
readable and writable."     fi     if [[ "$string" =~ ^[A-Z][a-z]+[0-9]+$
]]; then         echo "String matches the complex pattern."
fi
```

50. Signal Trapping * Task: Trap Ctrl-C for graceful exit. * Solution:

```
bash
#!/bin/bash     cleanup() {         echo "Caught interrupt signal.
Cleaning up..."         # Remove temp files, close connections,
etc.         exit 1     }     trap cleanup INT TERM     echo
"Running... Press Ctrl-C to test."     sleep 10
```

51. Here Documents * Task: Generate a dynamic file from variables. * Solution:

```
bash     #!/bin/bash     db_host="localhost"     db_user="admin"
cat > config.yml <<EOF     database:       host: $db_host       user:
```

```
$db_user       port: 3306     app:       name: "My App"     EOF
# To avoid variable expansion, use 'EOF'     cat > template.sql
<<'EOF'     SELECT * FROM users WHERE id = $user_id; -- $user_id
won't be expanded here     EOF
```

52. Process Substitution * Task: Compare output of two commands without temp files. * Solution: `diff <(ls -1 /dir1) <(ls -1 /dir2)` * Each `<(command)` creates a FIFO (named pipe) that `diff` reads from.

53. Debugging * Task: Demonstrate `set -x` and `trap DEBUG`. * Solution: `bash
#!/bin/bash   # Method 1: Simple tracing   set -x   echo
"This will be traced"   set +x   # Method 2: Custom DEBUG
trap   trap 'echo "DEBUG: executing: $BASH_COMMAND"' DEBUG
echo "Hello"   x=10   echo "World"   trap - DEBUG # Turn
off the trap`

54. The **exec** Command * Task: Demonstrate replacing the shell process. * Solution: `bash   #!/bin/bash   echo "This script is about
to replace itself with the 'top' command."   exec top   #
This line will never be executed, because the shell running this
script is now top.   echo "Goodbye!"`

55. Read a file line by line * Task: The most robust method to read a file. * Solution: `bash   while IFS= read -r line; do   # Process the
line, stored in $line   printf 'Processed: %s\n' "$line"
done < "input_file.txt"` * `IFS=`: Prevents leading/trailing whitespace from being trimmed. * `-r`: Prevents backslash escapes from being interpreted. * This is the gold standard for robustness.