

I am a linux administrator and want to learn the below technologies to grow in my work to the next level . In what order should i learn and how can you help me in learning 1 RHCE/Ansible 2 Azure admin certification 3 Kubernetes docker 4 Python 5 terraform 6 ci/cd 7 Monitoring elkstack prometheus grafana 8 bash scripting

12:42 PM

Great set of technologies! As a Linux admin, you already have a strong foundation. Here's the recommended learning order based on logical progression and how each skill builds on the previous:

Recommended Order:

- 1. Bash Scripting** — You likely know some already, but mastering it first sharpens your automation thinking and makes everything else easier. It's your daily tool.
- 2. Python** — Once you're comfortable scripting in Bash, Python opens up more powerful automation. It's also essential for Ansible and many DevOps tools.
- 3. RHCE / Ansible** — With Bash and Python under your belt, Ansible becomes much more intuitive. This is a natural evolution for a Linux admin and gives you your first real taste of Infrastructure as Code.
- 4. Docker / Kubernetes** — Start with Docker to understand containers, then move to Kubernetes for orchestration. This is the biggest mindset shift from traditional Linux admin work, so having solid scripting skills first helps a lot.
- 5. CI/CD** — Once you understand containers and automation, CI/CD pipelines (Jenkins, GitHub Actions, GitLab CI) make complete sense. You'll be building and deploying containerized apps through pipelines.
- 6. Terraform** — With CI/CD and containers in place, Terraform fits naturally as your cloud infrastructure provisioning tool. It pairs perfectly with the next step.
- 7. Azure Admin Certification** — By this point you'll already be working with cloud concepts through Terraform and CI/CD. Getting the Azure admin cert now will consolidate your cloud knowledge formally and add serious weight to your resume.
- 8. Monitoring (ELK Stack, Prometheus, Grafana)** — Best learned last because now you have real infrastructure to monitor. You'll understand *what* to monitor and *why*, which makes it far more meaningful.

How I can help you:

For each technology, I can help you with concept explanations from scratch with real-world Linux admin context, hands-on labs and practice exercises tailored to your background, interview prep and exam prep (especially for RHCE and Azure), writing and reviewing your Ansible playbooks, Python scripts, Terraform configs, Bash scripts, debugging errors you hit along the way, building real projects like a full monitoring stack or a CI/CD pipeline, and creating a structured study plan with timelines for each topic.

The best way to use me is to pick one topic, tell me your current level, and we go deep together. Where would you like to start?

We can start with the first topic bash scripting

12:43 PM

Great choice! Let's start with a quick assessment so I can tailor the learning to your level.

What's your current experience with Bash scripting?

Q: What is your current Bash scripting level? A: I know basic commands

Q: What is your main goal with Bash scripting? A: Automate sysadmin tasks, Prepare for RHCE, Build a strong base for Python/Ansible and Write production-grade scripts

12:44 PM

Perfect! You have a solid starting point with basic commands, and your goals are very well-rounded. We'll build you up from the fundamentals to production-grade scripting.

Here's the complete curriculum I've designed for you:

Your Bash Scripting Learning Path

Module 1 — Script Foundations Understanding shebangs, script structure, variables, user input, and running scripts properly. This is where good habits are built.

Module 2 — Conditionals & Logic if/else, case statements, test operators for files, strings, and numbers. The backbone of any real sysadmin script.

Module 3 — Loops for, while, until loops. Iterating over files, users, servers, logs — the bread and butter of automation.

Module 4 — Functions Writing reusable, clean functions. This is what separates a script from production-grade code.

Module 5 — Working with Files & Text grep, awk, sed, cut, sort inside scripts. Essential for log parsing and sysadmin tasks.

Module 6 — Error Handling & Debugging Exit codes, trap, set -e, set -x. Critical for writing scripts that are safe to run in production.

Module 7 — Real-World Sysadmin Projects Disk usage alerting, user creation automation, log rotation, backup scripts, system health reports. This ties everything together.

Each module will have a concept explanation, real examples relevant to your Linux admin work, and a hands-on exercise for you to try.

Shall we dive into **Module 1 — Script Foundations** right now?

Yes

12:44 PM

Module 1 — Script Foundations

Let's build the right habits from the start. Every production script you write will follow this structure.

1. The Shebang Line

The very first line of every bash script must be:

bash

#!/bin/bash

This tells the system *which interpreter* to use to run the script. Without it, the script may behave unpredictably depending on the shell the user is running.

2. Basic Script Structure

Here's what a well-structured bash script looks like:

bash

#!/bin/bash

=====

Script Name: system_info.sh

Description: Displays basic system information

Author: Your Name

Date: 2024-01-01

Version: 1.0

=====

--- Variables ---

HOSTNAME=\$(hostname)

OS=\$(cat /etc/os-release | grep PRETTY_NAME | cut -d= -f2)

UPTIME=\$(uptime -p)

DISK=\$(df -h / | awk 'NR==2 {print \$5}')

MEMORY=\$(free -h | awk '/Mem:/ {print \$3 "/" \$2}')

```
# --- Main Logic ---  
  
echo "===== System Information ====="  
echo "Hostname : $HOSTNAME"  
echo "OS      : $OS"  
echo "Uptime   : $UPTIME"  
echo "Disk Used : $DISK"  
echo "Memory   : $MEMORY"  
echo "=====
```

3. Variables — The Rules

bash

```
# Defining variables — NO spaces around =  
NAME="LinuxAdmin"  
AGE=5
```

```
# Accessing variables — always use $  
echo $NAME  
echo "I have $AGE years of experience"
```

```
# Command output stored in a variable  
CURRENT_USER=$(whoami)  
echo "Running as: $CURRENT_USER"
```

```
# Read-only variable (like a constant)  
readonly MAX_RETRIES=3
```

```
# Best practice — use UPPERCASE for your variable names  
# so they stand out clearly in your scripts
```

 Common mistake: Never put spaces around = when assigning variables. NAME = "test" will fail.

4. Taking User Input

bash

```
#!/bin/bash
```

```
# Basic input
```

```
read -p "Enter username to create: " USERNAME  
echo "You entered: $USERNAME"
```

```
# Silent input (for passwords)
```

```
read -sp "Enter password: " PASSWORD  
echo "" # move to new line after silent input
```

```
# Input with a timeout (useful in automated scripts)
```

```
read -t 10 -p "Enter your choice (10 sec timeout): " CHOICE
```

5. Making a Script Executable & Running It

bash

```
# Give execute permission
```

```
chmod +x system_info.sh
```

```
# Run it
```

```
./system_info.sh
```

```
# Or run with bash explicitly
```

```
bash system_info.sh
```

```
# Run as root if needed
```

```
sudo ./system_info.sh
```

6. Special Built-in Variables

These are used heavily in real sysadmin scripts:

bash

```
$0 # Name of the script itself  
$1 # First argument passed to the script  
$2 # Second argument  
$# # Total number of arguments passed  
$@ # All arguments  
$? # Exit code of the last command (0 = success)  
$$ # PID of the current script  
$USER # Current user running the script
```

Example using arguments:

bash

```
#!/bin/bash  
  
# Usage: ./create_user.sh john
```

```
USERNAME=$1
```

```
if [ -z "$USERNAME" ]; then  
    echo "Usage: $0 <username>"  
    exit 1  
fi
```

```
echo "Creating user: $USERNAME"  
useradd $USERNAME  
echo "Done!"
```

Your Exercise — Module 1

Try writing this script on your Linux machine:

Task: Write a script called server_info.sh that:

1. Accepts a name as an argument (e.g. ./server_info.sh John)
 2. Greets the user by name
 3. Displays: hostname, current logged-in user, current date & time, and how much disk space is used on /
 4. If no name is passed, it should print a usage message and exit
-

Give it a try and paste your script here. I'll review it, give feedback, and then we move to **Module 2 — Conditionals & Logic.** 

Yes

12:45 PM

It looks like you may have meant to paste your script but only "Yes" came through.

Go ahead and paste the script you wrote for the exercise and I'll review it in detail. If you'd like, I can also give you a hint or two to help you get started writing it!

What would you prefer?

Q: How would you like to proceed? A: Show me the solution and explain it

12:45 PM

Sure! Here's the complete solution with a detailed explanation of every part:

Solution — server_info.sh

```
bash  
#!/bin/bash  
# ======  
# Script Name: server_info.sh  
# Description: Displays server information  
# Usage   : ./server_info.sh <your_name>  
# ======
```

```

# --- Check if an argument was passed ---
if [ -z "$1" ]; then
    echo "Usage: $0 <your_name>"
    exit 1
fi

# --- Variables ---
NAME=$1
HOSTNAME=$(hostname)
CURRENT_USER=$(whoami)
DATE_TIME=$(date "+%Y-%m-%d %H:%M:%S")
DISK_USED=$(df -h / | awk 'NR==2 {print $3 " used out of " $2 " (" $5 ")"})'

```

```

# --- Main Output ---
echo "=====
echo " Welcome, $NAME!"
echo "=====
echo "Hostname : $HOSTNAME"
echo "Logged in as : $CURRENT_USER"
echo "Date & Time : $DATE_TIME"
echo "Disk Usage : $DISK_USED"
echo "====="

```

Line by Line Explanation

The shebang

bash

#!/bin/bash

Always the first line. Tells the OS to use bash to interpret this script.

Checking if an argument was passed

```
bash

if [ -z "$1" ]; then
    echo "Usage: $0 <your_name>"
    exit 1
fi
```

-z checks if a string is **empty**. So if the user runs the script without a name, it prints a usage message and exits with code 1 (meaning failure). \$0 prints the script name itself in the usage message which is a nice professional touch.

Storing the argument in a named variable

```
bash

NAME=$1
```

Instead of using \$1 everywhere, we store it in a meaningful variable name. This makes the script readable and easier to maintain.

Capturing command output into variables

```
bash

HOSTNAME=$(hostname)
CURRENT_USER=$(whoami)
DATE_TIME=$(date "+%Y-%m-%d %H:%M:%S")
```

The \$() syntax runs the command inside and stores its output. date "+%Y-%m-%d %H:%M:%S" formats the date in a clean readable way.

Parsing disk usage with awk

```
bash

DISK_USED=$(df -h / | awk 'NR==2 {print $3 " used out of " $2 " (" $5 ")"}')
````
```

`df -h /` shows disk usage of the root partition. The output looks like this:

```
```
```

```
Filesystem  Size  Used Avail Use% Mounted on
/dev/sda1    50G  12G  38G  24% /
```

awk 'NR==2' picks the **second line** (the data line, not the header). Then \$2, \$3, \$5 pick the Size, Used, and Use% columns respectively.

How to Run It

```
bash

# Give execute permission

chmod +x server_info.sh
```

Run with your name

```
./server_info.sh John
```

Run without argument to test the usage message

```
./server_info.sh
```

Expected Output

```
=====
```

Welcome, John!

```
=====
```

Hostname : prod-server-01

Logged in as : john

Date & Time : 2024-01-15 10:35:22

Disk Usage : 12G used out of 50G (24%)

```
=====
```

Key Takeaways from Module 1

- Always start with `#!/bin/bash` and a header comment block
 - Use `$1, $2` for arguments and always validate them
 - Use `$()` to capture command output into variables
 - Use meaningful variable names in UPPERCASE
 - Always provide a usage message when arguments are missing
 - Exit with code 1 on errors and 0 on success
-

Try running this on your machine and see the output. Once you're comfortable, we'll move to **Module 2 — Conditionals & Logic** where things start getting really powerful for sysadmin automation! Ready? 

Move

12:46 PM

Module 2 — Conditionals & Logic

This is where your scripts start making **decisions** — the core of any real sysadmin automation.

1. The if/elif/else Structure

bash

#!/bin/bash

```
# Basic syntax
if [ condition ]; then
    # do something
elif [ another_condition ]; then
    # do something else
else
    # fallback
fi
```

 Always have **spaces** inside the brackets [condition]. Missing spaces will cause errors.

2. File & Directory Test Operators

These are used constantly in sysadmin scripts:

bash

```
[ -f /etc/passwd ]    # True if file EXISTS  
[ -d /var/log ]      # True if DIRECTORY exists  
[ -r /etc/shadow ]   # True if file is READABLE  
[ -w /tmp/test ]     # True if file is WRITABLE  
[ -x /usr/bin/python ] # True if file is EXECUTABLE  
[ -s /var/log/syslog ] # True if file is NOT EMPTY  
[ -e /etc/hosts ]     # True if path EXISTS (file or dir)
```

Real example:

bash

```
#!/bin/bash
```

```
LOGFILE="/var/log/app.log"
```

```
if [ -f "$LOGFILE" ]; then  
    echo "Log file found. Size: $(du -sh $LOGFILE | cut -f1)"  
else  
    echo "Log file not found. Creating it..."  
    touch "$LOGFILE"  
    echo "Log file created successfully."  
fi
```

3. String Test Operators

bash

```
[ -z "$VAR" ]      # True if string is EMPTY  
[ -n "$VAR" ]      # True if string is NOT empty
```

```
[ "$A" = "$B" ]      # True if strings are EQUAL  
[ "$A" != "$B" ]     # True if strings are NOT equal
```

Real example:

bash

```
#!/bin/bash
```

```
SERVICE=$1
```

```
if [ -z "$SERVICE" ]; then  
    echo "Usage: $0 <service_name>"  
    exit 1  
fi
```

```
STATUS=$(systemctl is-active $SERVICE)
```

```
if [ "$STATUS" = "active" ]; then  
    echo "✓ $SERVICE is running"  
else  
    echo "✗ $SERVICE is NOT running. Attempting restart..."  
    systemctl restart $SERVICE  
    echo "Restart triggered for $SERVICE"  
fi
```

4. Numeric Test Operators

bash

```
[ $A -eq $B ]  # Equal  
[ $A -ne $B ]  # Not equal  
[ $A -gt $B ]  # Greater than  
[ $A -lt $B ]  # Less than
```

```
[ $A -ge $B ] # Greater than or equal
```

```
[ $A -le $B ] # Less than or equal
```

Real example — Disk usage alert:

```
bash
```

```
#!/bin/bash
```

```
THRESHOLD=80
```

```
DISK_USAGE=$(df / | awk 'NR==2 {print $5}' | tr -d '%')
```

```
if [ $DISK_USAGE -ge $THRESHOLD ]; then
```

```
    echo "⚠️ WARNING: Disk usage is at ${DISK_USAGE}% — above threshold of ${THRESHOLD}%"
```

```
else
```

```
    echo "✓ Disk usage is at ${DISK_USAGE}% — within safe limits"
```

```
fi
```

5. Combining Conditions

```
bash
```

```
# AND — both conditions must be true
```

```
if [ $DISK -gt 80 ] && [ -f "/etc/alert.conf" ]; then
```

```
    echo "Alert config exists and disk is high"
```

```
fi
```

```
# OR — at least one condition must be true
```

```
if [ "$USER" = "root" ] || [ "$USER" = "admin" ]; then
```

```
    echo "Privileged user detected"
```

```
fi
```

6. The case Statement

Great alternative to long if/elif chains — perfect for menus and argument handling:

```

bash

#!/bin/bash

ACTION=$1

case "$ACTION" in
    start)
        echo "Starting the service..."
        systemctl start nginx
        ;;
    stop)
        echo "Stopping the service..."
        systemctl stop nginx
        ;;
    restart)
        echo "Restarting the service..."
        systemctl restart nginx
        ;;
    status)
        systemctl status nginx
        ;;
    *)
        echo "Usage: $0 {start|stop|restart|status}"
        exit 1
        ;;
esac

```

The *) at the end is the **default case** — like else. Always include it.

7. Checking if Running as Root

This is something you'll put in almost every production sysadmin script:

bash

#!/bin/bash

```
if [ "$EUID" -ne 0 ]; then
    echo "✖ This script must be run as root. Use sudo."
    exit 1
fi
```

echo "✓ Running as root. Proceeding..."

\$EUID is the **effective user ID**. Root always has EUID of 0.

Your Exercise — Module 2

Write a script called service_check.sh that:

1. Checks if it is being run as root — if not, exit with a message
 2. Accepts a service name as an argument — if none given, show usage and exit
 3. Checks if the service is **active** — if yes, print a success message
 4. If the service is **not active**, attempt to restart it
 5. After restarting, check again — if it's now active print success, if still failing print a critical alert
 6. As a bonus — check if disk usage on / is above 85% and warn the user at the end
-

This exercise combines everything from Module 1 and Module 2. Give it a try, paste your script here, or let me know if you want the solution explained! 

Provide the script and explain then move to module 3

12:47 PM

Solution — service_check.sh

bash

#!/bin/bash

```
# =====
```

```
# Script Name: service_check.sh

# Description: Checks and recovers a service,
#               and monitors disk usage

# Usage   : sudo ./service_check.sh <service_name>
# =====

# --- Check if running as root ---

if [ "$EUID" -ne 0 ]; then
    echo "✖ This script must be run as root. Use sudo."
    exit 1
fi

# --- Check if argument was passed ---

if [ -z "$1" ]; then
    echo "Usage: $0 <service_name>"
    exit 1
fi

# --- Variables ---

SERVICE=$1
DISK_THRESHOLD=85
DISK_USAGE=$(df / | awk 'NR==2 {print $5}' | tr -d '%')

# --- Check service status ---

STATUS=$(systemctl is-active $SERVICE)

if [ "$STATUS" = "active" ]; then
    echo "✓ $SERVICE is running normally."
else
```

```

echo "⚠️ $SERVICE is NOT running. Attempting restart..."

systemctl restart $SERVICE


# --- Re-check after restart ---

STATUS=$(systemctl is-active $SERVICE)

if [ "$STATUS" = "active" ]; then
    echo "✓ $SERVICE successfully restarted."
else
    echo "❗ CRITICAL: $SERVICE failed to restart. Manual intervention required!"
    exit 1
fi

# --- Bonus: Disk usage check ---

if [ $DISK_USAGE -ge $DISK_THRESHOLD ]; then
    echo "⚠️ WARNING: Disk usage is at ${DISK_USAGE}% — above ${DISK_THRESHOLD}% threshold!"
else
    echo "✓ Disk usage is at ${DISK_USAGE}% — within safe limits."
fi

```

Explanation

Root check

bash

```
if [ "$EUID" -ne 0 ]; then
```

\$EUID is the effective user ID. Root is always 0. If someone runs without sudo, the script exits immediately before doing anything dangerous.

Argument validation

```
bash  
if [ -z "$1" ]; then
```

-z checks if the argument is empty. If no service name is passed, we print usage and exit cleanly rather than letting the script crash midway.

Capturing service status

```
bash  
STATUS=$(systemctl is-active $SERVICE)  
systemctl is-active returns the word active if running, or inactive / failed otherwise. We store it in a variable and compare it as a string.
```

Restart and re-check logic

```
bash  
systemctl restart $SERVICE  
STATUS=$(systemctl is-active $SERVICE)
```

After attempting a restart we immediately re-check the status. This is the right pattern in production — never assume a restart succeeded. Always verify.

Disk check with tr

```
bash  
DISK_USAGE=$(df / | awk 'NR==2 {print $5}' | tr -d '%')  
df / gives disk info. awk 'NR==2 {print $5}' grabs the usage percentage column. tr -d '%' strips the % sign so we can use it in numeric comparisons.
```

How to Run It

```
bash  
chmod +x service_check.sh  
  
# Check nginx  
sudo ./service_check.sh nginx
```

```
# Check sshd  
sudo ./service_check.sh sshd
```

```
# Test with no argument  
sudo ./service_check.sh
```

...

```
## 📄 Expected Output
```

...

✓ sshd is running normally.

⚠️ WARNING: Disk usage is at 87% — above 85% threshold!

Module 3 — Loops

Loops let you automate repetitive tasks across multiple files, users, servers, or log entries. This is where bash scripting becomes truly powerful for sysadmins.

1. The for Loop

bash

```
# Basic syntax  
for VARIABLE in list; do  
    # do something  
done
```

Loop over a simple list:

bash

```
#!/bin/bash
```

```
for SERVICE in nginx sshd cron firewalld; do
    STATUS=$(systemctl is-active $SERVICE)
    echo "$SERVICE : $STATUS"
done
```

Loop over files:

```
bash
#!/bin/bash
```

```
for LOGFILE in /var/log/*.log; do
    SIZE=$(du -sh $LOGFILE | cut -f1)
    echo "$LOGFILE — Size: $SIZE"
```

```
done
```

Loop over command output:

```
bash
#!/bin/bash
```

```
# Check disk usage for all mounted filesystems
for MOUNT in $(df -h | awk 'NR>1 {print $6}'); do
    echo "Checking: $MOUNT"
done
```

2. The while Loop

Runs as long as a condition is **true**. Great for monitoring and retry logic.

```
bash
# Basic syntax
while [ condition ]; do
    # do something
done
```

Real example — retry logic:

```
bash

#!/bin/bash

SERVICE="nginx"
MAX_RETRIES=3
COUNT=0

while [ $COUNT -lt $MAX_RETRIES ]; do
    STATUS=$(systemctl is-active $SERVICE)

    if [ "$STATUS" = "active" ]; then
        echo "✓ $SERVICE is running."
        break # exit the loop immediately
    else
        COUNT=$((COUNT + 1))
        echo "Attempt $COUNT: $SERVICE not running. Retrying in 5 seconds..."
        sleep 5
        systemctl restart $SERVICE
    fi
done
```

```
if [ $COUNT -eq $MAX_RETRIES ]; then
    echo "⚠ CRITICAL: $SERVICE failed after $MAX_RETRIES attempts!"
fi
```

Reading a file line by line:

```
bash

#!/bin/bash
```

```
# Process a list of servers from a file
```

```
while IFS= read -r SERVER; do
    echo "Pinging $SERVER..."
    ping -c 1 $SERVER &>/dev/null && echo "✓ $SERVER is UP" || echo "✗ $SERVER is DOWN"
done < /etc/serverlist.txt
```

This pattern — reading a file line by line — is one of the most used patterns in real sysadmin scripting.

3. The until Loop

Opposite of while — runs until a condition becomes **true**.

bash

```
#!/bin/bash
```

```
# Wait until a service comes up
SERVICE="nginx"

until [ "$(systemctl is-active $SERVICE)" = "active" ]; do
    echo "Waiting for $SERVICE to start..."
    sleep 3
done

echo "✓ $SERVICE is now active!"
```

This is perfect for startup scripts where you need to wait for a dependency to be ready before proceeding.

4. Loop Control — break and continue

bash

```
# break — exits the loop entirely
# continue — skips current iteration and moves to next

#!/bin/bash
```

```
for USER in john mary root guest admin; do
    # Skip root account
    if [ "$USER" = "root" ]; then
        echo "Skipping root..."
        continue
    fi

    # Stop if we hit guest
    if [ "$USER" = "guest" ]; then
        echo "Guest found. Stopping."
        break
    fi

    echo "Processing user: $USER"
done
```

5. Arithmetic in Loops

```
bash
#!/bin/bash
```

```
# C-style for loop
for (( i=1; i<=5; i++ )); do
    echo "Iteration: $i"
done
```

```
# Countdown
COUNT=10
while [ $COUNT -gt 0 ]; do
```

```
echo "Countdown: $COUNT"
COUNT=$((COUNT - 1))
sleep 1
done
echo "🚀 Launch!"
```

6. Real-World Example — Multi-Server Health Check

This is the kind of script you'd actually use in production:

```
bash
#!/bin/bash

# =====
# Script Name: health_check.sh
# Description: Checks health of multiple servers
# Usage   : ./health_check.sh
# =====

SERVERS=("web01" "web02" "db01" "db02" "cache01")
SERVICES=("nginx" "sshd" "cron")
REPORT="/tmp/health_report_$(date +%Y%m%d).txt"

echo "===== Server Health Report — $(date) =====" | tee $REPORT

for SERVER in "${SERVERS[@]}"; do
    echo "" | tee -a $REPORT
    echo "--- Checking: $SERVER ---" | tee -a $REPORT

    # Ping check
    ping -c 1 $SERVER &>/dev/null
    if [ $? -eq 0 ]; then
```

```

echo "✓ $SERVER is reachable" | tee -a $REPORT

else

echo "✗ $SERVER is UNREACHABLE" | tee -a $REPORT

continue # skip service check if server is down

fi

# Service check via SSH

for SERVICE in "${SERVICES[@]}"; do

STATUS=$(ssh -o ConnectTimeout=5 $SERVER "systemctl is-active $SERVICE" 2>/dev/null)

if [ "$STATUS" = "active" ]; then

echo " ✓ $SERVICE is running" | tee -a $REPORT

else

echo " ✗ $SERVICE is NOT running" | tee -a $REPORT

fi

done

done

echo "" | tee -a $REPORT

echo "===== Report saved to $REPORT =====" | tee -a $REPORT

```

Key Takeaways from Module 3

- Use for when you know the list of items upfront — files, users, services, servers
 - Use while for retry logic, monitoring loops, and reading files line by line
 - Use until when waiting for something to become ready
 - Always use break and continue to control loop flow cleanly
 - Arrays ("item1" "item2") with "\${ARRAY[@]}" are powerful for server/service lists
 - tee lets you print to screen and write to a file at the same time — great for reports
-

Your Exercise — Module 3

Write a script called user_audit.sh that:

1. Loops through all users in /etc/passwd with a shell of /bin/bash (real users)
 2. For each user, checks if their home directory exists
 3. Checks if the home directory is empty or has files in it
 4. Prints a clean report showing each user, their home directory status, and file count
 5. At the end prints a summary of total users checked
-

Paste your attempt here or let me know if you'd like the solution and we'll move straight to **Module 4 — Functions!** 

Provide the script and explain then move to next module

12:48 PM

Solution — user_audit.sh

bash

```
#!/bin/bash

# =====

# Script Name: user_audit.sh

# Description: Audits real users and their
#               home directory status

# Usage   : ./user_audit.sh

# =====

# --- Variables ---

REPORT="/tmp/user_audit_$(date +%Y%m%d).txt"

TOTAL=0

MISSING=0

EMPTY=0

POPULATED=0
```

```

echo "===== User Audit Report — $(date) =====" | tee $REPORT
echo "" | tee -a $REPORT

# --- Loop through users with /bin/bash shell ---
while IFS=: read -r USERNAME PASSWORD UID GID COMMENT HOME SHELL; do
    if [ "$SHELL" = "/bin/bash" ]; then
        TOTAL=$((TOTAL + 1))

        echo "User: $USERNAME" | tee -a $REPORT
        echo " Home: $HOME" | tee -a $REPORT

        # Check if home directory exists
        if [ ! -d "$HOME" ]; then
            echo " Status: ❌ Home directory MISSING" | tee -a $REPORT
            MISSING=$((MISSING + 1))
        else
            # Count files in home directory
            FILE_COUNT=$(ls -A "$HOME" | wc -l)

            if [ "$FILE_COUNT" -eq 0 ]; then
                echo " Status: ⚠️ Home directory is EMPTY" | tee -a $REPORT
                EMPTY=$((EMPTY + 1))
            else
                echo " Status: ✓ Home directory has $FILE_COUNT file(s)" | tee -a $REPORT
                POPULATED=$((POPULATED + 1))
            fi
        fi
        echo "" | tee -a $REPORT
    fi
done

```

```
done < /etc/passwd

# --- Summary ---

echo "===== Summary =====" | tee -a $REPORT
echo "Total Users Checked : $TOTAL" | tee -a $REPORT
echo "Populated Homes   : $POPULATED" | tee -a $REPORT
echo "Empty Homes       : $EMPTY" | tee -a $REPORT
echo "Missing Homes     : $MISSING" | tee -a $REPORT
echo "Report saved to   : $REPORT" | tee -a $REPORT
```

Explanation

Reading /etc/passwd cleanly

bash

```
while IFS=: read -r USERNAME PASSWORD UID GID COMMENT HOME SHELL; do
```

/etc/passwd is colon-separated. Setting IFS=: tells bash to split each line by : and map each field into a named variable. This is far cleaner than using cut or awk to extract fields manually.

Filtering only real users

bash

```
if [ "$SHELL" = "/bin/bash" ]; then
```

System accounts like daemon, nobody, and www-data use /sbin/nologin or /bin/false as their shell.
Filtering by /bin/bash gives us only real human users.

Counting files in a directory

bash

```
FILE_COUNT=$(ls -A "$HOME" | wc -l)
```

ls -A lists all files including hidden ones but excludes . and ... Piping to wc -l counts the lines which equals the number of files.

Counters using arithmetic

```
bash
```

```
TOTAL=$((TOTAL + 1))
```

`$(())` is bash's arithmetic operator. We increment counters for each category so we can print a meaningful summary at the end.

tee for simultaneous output

```
bash
```

```
echo "..." | tee -a $REPORT
```

```
...
```

`tee` prints to the screen AND writes to the file at the same time. `-a` means append so we don't overwrite previous lines.

```
---
```

```
## 📄 Expected Output
```

```
...
```

```
===== User Audit Report — 2024-01-15 =====
```

User: root

Home: /root

Status: ✓ Home directory has 12 file(s)

User: john

Home: /home/john

Status: ✓ Home directory has 45 file(s)

User: mary

Home: /home/mary

Status: ! Home directory is EMPTY

```
===== Summary =====
```

```
Total Users Checked : 3
```

```
Populated Homes : 2
```

```
Empty Homes : 1
```

```
Missing Homes : 0
```

```
Report saved to : /tmp/user_audit_20240115.txt
```

Module 4 — Functions

Functions are what transform a basic script into clean, maintainable, production-grade code. They allow you to write logic once and reuse it anywhere in your script.

1. Basic Function Syntax

```
bash
```

```
# Define a function
```

```
function_name() {
```

```
    # code here
```

```
}
```

```
# Call a function
```

```
function_name
```

Simple example:

```
bash
```

```
#!/bin/bash
```

```
# Define
```

```
greet_user() {
```

```
    echo "Hello, $1! Welcome to the server."
```

```
}
```

```
# Call  
greet_user "John"  
greet_user "Mary"
```

 Always **define** functions before you **call** them. Bash reads top to bottom.

2. Functions with Arguments

Functions receive arguments the same way scripts do — via \$1, \$2, etc:

bash

```
#!/bin/bash
```

```
check_service() {  
    SERVICE=$1  
  
    STATUS=$(systemctl is-active $SERVICE)  
  
    if [ "$STATUS" = "active" ]; then  
        echo "✓ $SERVICE is running"  
    else  
        echo "✗ $SERVICE is NOT running"  
    fi  
}
```

```
# Call the function for multiple services  
check_service nginx  
check_service sshd  
check_service cron
```

3. Return Values & Exit Codes

Functions can return a status code using `return`. Use `$?` to capture it:

bash

```
#!/bin/bash
```

```
is_service_running() {  
    SERVICE=$1  
    STATUS=$(systemctl is-active $SERVICE 2>/dev/null)  
  
    if [ "$STATUS" = "active" ]; then  
        return 0 # success  
    else  
        return 1 # failure  
    fi  
}
```

Use the return value

```
if is_service_running nginx; then  
    echo "✓ nginx is UP"  
else  
    echo "✗ nginx is DOWN"  
fi
```

return only passes a number (0–255). To return actual text from a function, use `echo` inside the function and capture it with `$()`.

Returning text from a function:

bash

```
#!/bin/bash
```

```
get_disk_usage() {  
    df / | awk 'NR==2 {print $5}' | tr -d '%'
```

```
}
```

```
# Capture the output  
USAGE=$(get_disk_usage)  
echo "Current disk usage: ${USAGE}%"
```

4. Local Variables

Without local, variables inside functions are **global** and can accidentally overwrite other variables.
Always use local inside functions:

bash

```
#!/bin/bash
```

```
get_user_count() {  
    local USER_COUNT=$(grep "/bin/bash" /etc/passwd | wc -l)  
    echo $USER_COUNT  
}
```

```
check_disk() {  
    local THRESHOLD=$1  
    local USAGE=$(df / | awk 'NR==2 {print $5}' | tr -d '%')
```

```
    if [ $USAGE -ge $THRESHOLD ]; then  
        echo "⚠ Disk at ${USAGE}% — above ${THRESHOLD}% threshold"  
        return 1  
    else  
        echo "✓ Disk at ${USAGE}% — within safe limits"  
        return 0  
    fi
```

```
# Call functions  
  
USERS=$(get_user_count)  
echo "Total bash users: $USERS"  
  
check_disk 80
```

5. A Logger Function — Used in Every Production Script

This is a pattern you'll use in almost every real script:

bash

```
#!/bin/bash
```

```
LOGFILE="/var/log/myscript.log"
```

```
# --- Logger function ---
```

```
log() {  
    local LEVEL=$1  
    local MESSAGE=$2  
    local TIMESTAMP=$(date "+%Y-%m-%d %H:%M:%S")  
    echo "[${TIMESTAMP}] [${LEVEL}] ${MESSAGE}" | tee -a ${LOGFILE}  
}
```

```
# --- Usage ---
```

```
log "INFO"  "Script started"  
log "INFO"  "Checking services..."  
log "WARNING" "Disk usage is high"  
log "ERROR"  "Service failed to start"  
log "INFO"  "Script completed"  
...
```

****Output:****

...

```
[2024-01-15 10:30:01] [INFO] Script started
[2024-01-15 10:30:01] [INFO] Checking services...
[2024-01-15 10:30:02] [WARNING] Disk usage is high
[2024-01-15 10:30:03] [ERROR] Service failed to start
[2024-01-15 10:30:03] [INFO] Script completed
```

6. Real-World Example — Full Structured Script with Functions

bash

```
#!/bin/bash

# =====

# Script Name: system_health.sh
# Description: Full system health check using
# modular functions
# Usage   : sudo ./system_health.sh
# =====

LOGFILE="/var/log/system_health.log"
DISK_THRESHOLD=85
SERVICES=("nginx" "sshd" "cron")

# --- Functions ---

log() {
    local LEVEL=$1
    local MESSAGE=$2
    local TIMESTAMP=$(date "+%Y-%m-%d %H:%M:%S")
    echo "[${TIMESTAMP}] [${LEVEL}] ${MESSAGE}" > ${LOGFILE}
}
```

```

echo "[${TIMESTAMP}] [${LEVEL}] ${MESSAGE}" | tee -a ${LOGFILE}
}

check_root() {
    if [ "$EUID" -ne 0 ]; then
        echo "✗ Run as root. Use sudo."
        exit 1
    fi
}

check_disk() {
    local USAGE=$(df / | awk 'NR==2 {print $5}' | tr -d '%')
    if [ $USAGE -ge ${DISK_THRESHOLD} ]; then
        log "WARNING" "Disk usage is at ${USAGE}% — above threshold!"
    else
        log "INFO" "Disk usage is at ${USAGE}% — OK"
    fi
}

check_memory() {
    local TOTAL=$(free -m | awk '/Mem:/ {print $2}')
    local USED=$(free -m | awk '/Mem:/ {print $3}')
    local PERCENT=$(( USED * 100 / TOTAL ))

    if [ $PERCENT -ge 90 ]; then
        log "WARNING" "Memory usage is at ${PERCENT}% — critical!"
    else
        log "INFO" "Memory usage is at ${PERCENT}% — OK"
    fi
}

```

```
}
```

```
check_services() {  
    for SERVICE in "${SERVICES[@]}"; do  
        if is_running $SERVICE; then  
            log "INFO" "$SERVICE is running"  
        else  
            log "ERROR" "$SERVICE is NOT running — attempting restart"  
            systemctl restart $SERVICE  
            if is_running $SERVICE; then  
                log "INFO" "$SERVICE restarted successfully"  
            else  
                log "ERROR" "$SERVICE failed to restart — manual intervention needed!"  
            fi  
        fi  
    done  
}
```

```
is_running() {  
    local SERVICE=$1  
    [ "$(systemctl is-active $SERVICE)" = "active" ]  
}
```

```
# --- Main ---  
check_root  
log "INFO" "===== Health Check Started ====="  
check_disk  
check_memory  
check_services
```

```
log "INFO" "===== Health Check Completed ====="
```

Key Takeaways from Module 4

- Always use local for variables inside functions to avoid side effects
 - Use return 0 for success and return 1 for failure — check with \$? or directly in if
 - Use echo inside a function and \$() outside to return text values
 - A log() function should be in every production script you write
 - Break your script into small focused functions — one function, one job
 - Keep your # --- Main --- section at the bottom clean and readable
-

Ready for **Module 5 — Working with Files & Text** where we master grep, awk, sed and cut inside scripts?



Yes

12:50 PM

Module 5 — Working with Files & Text

As a sysadmin, most of your automation will involve parsing logs, config files, and command outputs. Mastering these tools makes you extremely powerful.

1. grep — Searching Text

bash

```
# Basic syntax
```

```
grep "pattern" filename
```

```
# Most useful flags
```

```
grep -i "error" /var/log/syslog      # Case insensitive
```

```
grep -r "failed" /var/log/          # Recursive search in directory
```

```
grep -v "INFO" /var/log/app.log    # Invert — show lines NOT matching
```

```
grep -n "error" /var/log/app.log   # Show line numbers
```

```
grep -c "error" /var/log/app.log   # Count matching lines
```

```
grep -l "error" /var/log/*.log      # Show only filenames that match
grep -A 3 "error" /var/log/app.log   # Show 3 lines AFTER match
grep -B 3 "error" /var/log/app.log   # Show 3 lines BEFORE match
grep -E "error|failed|critical" file # Extended regex — match multiple patterns
```

Real sysadmin examples:

bash

```
#!/bin/bash
```

```
LOGFILE="/var/log/auth.log"
```

```
# Find all failed SSH login attempts
echo "==== Failed SSH Logins ===="
grep "Failed password" $LOGFILE | awk '{print $11}' | sort | uniq -c | sort -rn
```

```
# Find which users had successful logins
echo "==== Successful Logins ===="
grep "Accepted password" $LOGFILE | awk '{print $9}' | sort | uniq -c
```

```
# Count today's errors
TODAY=$(date +"%b %d")
ERROR_COUNT=$(grep "$TODAY" $LOGFILE | grep -c "error")
echo "Errors today: $ERROR_COUNT"
```

2. awk — Column & Field Processing

awk is your go-to tool when you need specific columns from structured output:

bash

```
# Basic syntax — process field number N
awk '{print $N}' filename
```

```

# Key built-in variables

$0 # Entire line

$1 # First field

$NF # Last field

NR # Current line number

NF # Number of fields in current line

FS # Field separator (default: space)

Common patterns:

bash

# Print specific columns from df output

df -h | awk '{print $1, $5}'

# Print lines where field 3 is greater than 80

df / | awk 'NR>1 && $5+0 > 80 {print $1, "is above 80%", $5}'

# Custom field separator — parse /etc/passwd

awk -F: '{print $1, $6}' /etc/passwd

# Print lines matching a pattern

awk '/error/ {print NR": "$0}' /var/log/app.log

# Sum a column — total memory of all processes

ps aux | awk '{sum += $6} END {print "Total Memory Used:", sum/1024, "MB"}'

# Print lines between two patterns

awk '/START/,/END/ {print}' logfile.txt

Real example — process report:

bash

#!/bin/bash

```

```
echo "==" Top 10 Memory Consuming Processes =="  
ps aux --sort=-%mem | awk 'NR<=11 {printf "%-10s %-8s %-8s %s\n", $1, $2, $4, $11}'  
  
echo ""  
echo "==" CPU Usage by User =="  
ps aux | awk 'NR>1 {cpu[$1]+=$3} END {for (user in cpu) print cpu[user]"%", user}' | sort -rn
```

3. sed — Stream Editor for Text Manipulation

sed is perfect for find-and-replace, deleting lines, and transforming text in files:

bash

```
# Basic syntax  
sed 'command' filename
```

Most used commands

```
sed 's/old/new/' file      # Replace FIRST occurrence per line  
sed 's/old/new/g' file     # Replace ALL occurrences per line  
sed 's/old/new/gi' file    # Replace all, case insensitive  
sed -i 's/old/new/g' file  # Edit file IN PLACE (modifies actual file)  
sed -i.bak 's/old/new/g' file # Edit in place but keep .bak backup  
sed '/pattern/d' file      # Delete lines matching pattern  
sed -n '5,10p' file        # Print only lines 5 to 10  
sed -n '/error/p' file      # Print only lines matching pattern
```

Real sysadmin examples:

bash

```
#!/bin/bash
```

```
CONFIG="/etc/nginx/nginx.conf"
```

```
# Backup before editing — always!
cp $CONFIG ${CONFIG}.bak

# Change worker processes value
sed -i 's/worker_processes auto/worker_processes 4/g' $CONFIG

# Comment out a line
sed -i 's/^listen 80/#listen 80/' $CONFIG

# Remove all blank lines from a file
sed -i '/^$/d' $CONFIG

# Remove comments from a config file
sed -i '/^#/d' $CONFIG

# Replace an IP address across a config
OLD_IP="192.168.1.10"
NEW_IP="192.168.1.20"
sed -i "s/$OLD_IP/$NEW_IP/g" $CONFIG

echo "Config updated successfully"
```

4. cut — Extract Specific Fields

Simpler than awk for basic column extraction:

```
bash
# Basic syntax
cut -d'DELIMITER' -fN filename

# Examples
```

```
cut -d: -f1 /etc/passwd      # Extract usernames  
cut -d: -f1,6 /etc/passwd    # Extract username and home dir  
cut -d',' -f2 report.csv    # Extract second column from CSV  
cut -c1-10 file.txt        # Extract first 10 characters of each line  
  
# Combined with other tools  
df -h | grep -v Filesystem | cut -c55-58  # Get usage % column
```

5. sort and uniq — Organizing Output

```
bash  
  
sort file.txt      # Sort alphabetically  
sort -r file.txt  # Reverse sort  
sort -n file.txt  # Sort numerically  
sort -k2 file.txt # Sort by second column  
sort -t: -k3 -n /etc/passwd # Sort passwd by UID  
  
uniq file.txt      # Remove consecutive duplicates  
uniq -c file.txt  # Count occurrences  
uniq -d file.txt  # Show only duplicates  
  
# The classic combo — count and rank occurrences  
cat /var/log/auth.log | grep "Failed" | awk '{print $11}' | sort | uniq -c | sort -rn
```

6. Real-World Example — Log Analysis Script

This is the kind of script that impresses in production environments:

```
bash  
  
#!/bin/bash  
# ======  
# Script Name: log_analyzer.sh
```

```
# Description: Analyzes auth log for security
#           insights and suspicious activity

# Usage    : sudo ./log_analyzer.sh
# =====

LOGFILE="/var/log/auth.log"
REPORT="/tmp/security_report_$(date +%Y%m%d_%H%M%S).txt"
THRESHOLD=10

# --- Logger ---
log() {
    local LEVEL=$1
    local MESSAGE=$2
    echo "[${LEVEL}] ${MESSAGE}" | tee -a $REPORT
}

# --- Check log file exists ---
if [ ! -f "$LOGFILE" ]; then
    echo "✖ Log file not found: $LOGFILE"
    exit 1
fi

echo "===== Security Report — $(date) =====" | tee $REPORT
echo "" | tee -a $REPORT

# --- Failed login attempts ---
log "INFO" "==== Failed SSH Login Attempts (Top 10 IPs) ===="
grep "Failed password" $LOGFILE \
    | awk '{print $11}' \
```

```
| sort \
| uniq -c \
| sort -rn \
| head -10 \
| while read COUNT IP; do
    if [ $COUNT -ge $THRESHOLD ]; then
        log "WARNING" "$IP — $COUNT attempts ⚠ POSSIBLE BRUTE FORCE"
    else
        log "INFO" "$IP — $COUNT attempts"
    fi
done

echo "" | tee -a $REPORT
```

```
# --- Successful logins ---
log "INFO" "==== Successful Logins ===="
grep "Accepted password" $LOGFILE \
    | awk '{print $9, "from", $11}' \
    | sort \
    | uniq -c \
    | sort -rn \
    | head -10 \
    | tee -a $REPORT
```

```
echo "" | tee -a $REPORT

# --- Invalid users ---
log "INFO" "==== Login Attempts with Invalid Users ===="
grep "Invalid user" $LOGFILE \
```

```

| awk '{print $8}' \
| sort \
| uniq -c \
| sort -rn \
| head -10 \
| tee -a $REPORT

echo "" | tee -a $REPORT

# --- Root login attempts ---

ROOT_ATTEMPTS=$(grep "Failed password for root" $LOGFILE | wc -l)
if [ $ROOT_ATTEMPTS -gt 0 ]; then
    log "WARNING" "==== $ROOT_ATTEMPTS Failed ROOT login attempts detected! ==="
else
    log "INFO" "==== No failed root login attempts ==="
fi

echo "" | tee -a $REPORT
echo "===== Report saved to: $REPORT =====" | tee -a $REPORT

```

Key Takeaways from Module 5

- Use grep to **find** lines matching a pattern in logs and files
 - Use awk to **extract and process columns** from structured output
 - Use sed to **find, replace, and transform** text in files — always backup with -i.bak
 - Use cut for **simple column extraction** when awk is overkill
 - Chain tools with pipes | to build powerful one-liners
 - sort | uniq -c | sort -rn is the most powerful combo for log analysis
 - Never use sed -i without a backup in production scripts
-

Your Exercise — Module 5

Write a script called disk_report.sh that:

1. Uses df and awk to list all mounted filesystems with usage above 20%
 2. Uses grep and awk to extract the top 5 largest files in /var/log
 3. Uses sed to generate a clean formatted report saved to /tmp/disk_report.txt replacing any % with the word percent in the output
 4. Uses sort to rank filesystems from highest to lowest usage
-

Paste your attempt or say "**show solution**" and we'll move straight to **Module 6 — Error Handling & Debugging!** 

Provide the script and explain then move to next module

12:51 PM

Solution — disk_report.sh

bash

```
#!/bin/bash

# =====

# Script Name: disk_report.sh

# Description: Generates a disk usage report

#       with top large files analysis

# Usage   : ./disk_report.sh

# =====
```

REPORT="/tmp/disk_report.txt"

THRESHOLD=20

```
# --- Header ---

echo "===== Disk Usage Report — $(date) =====" | tee $REPORT

echo "" | tee -a $REPORT
```

```

# --- Section 1: Filesystems above threshold ---

echo "==== Filesystems Above ${THRESHOLD}% Usage (Ranked) ===" | tee -a $REPORT
df -h | awk 'NR>1 {print $5, $1, $3, $2, $6}' \
| tr -d '%' \
| sort -rn \
| while read USAGE FS USED SIZE MOUNT; do
if [ "$USAGE" -ge "$THRESHOLD" ]; then
    echo "${USAGE} percent used — $FS mounted on $MOUNT ($USED of $SIZE)" | tee -a $REPORT
fi
done

echo "" | tee -a $REPORT

# --- Section 2: Top 5 largest files in /var/log ---

echo "==== Top 5 Largest Files in /var/log ===" | tee -a $REPORT
find /var/log -type f -exec du -sh {} + 2>/dev/null \
| sort -rh \
| head -5 \
| awk '{print $1, $2}' \
| tee -a $REPORT

echo "" | tee -a $REPORT

# --- Section 3: Clean up % signs in report using sed ---

sed -i 's/%/ percent/g' $REPORT

echo "===== Report saved to: $REPORT =====" | tee -a $REPORT

```



Explanation

Extracting and sorting filesystem usage

bash

```
df -h | awk 'NR>1 {print $5, $1, $3, $2, $6}' \  
| tr -d '%' \  
| sort -rn
```

df -h gives human-readable disk info. awk 'NR>1' skips the header line and reorders columns putting usage percentage first so sort -rn can rank numerically from highest to lowest. tr -d '%' strips the % so the number can be used in arithmetic comparison.

Reading the sorted output into variables

bash

```
while read USAGE FS USED SIZE MOUNT; do  
    if [ "$USAGE" -ge "$THRESHOLD" ]; then
```

Piping into a while read loop lets us assign each column to a named variable cleanly. We then compare USAGE numerically against our threshold and only print filesystems that exceed it.

Finding largest files with find and du

bash

```
find /var/log -type f -exec du -sh {} + 2>/dev/null \  
| sort -rh \  
| head -5
```

find /var/log -type f finds all files recursively. -exec du -sh {} + runs du on all found files in one efficient call. sort -rh sorts in reverse human-readable order so 1G ranks above 500M. head -5 limits to top 5. 2>/dev/null suppresses permission denied errors cleanly.

Replacing % signs with sed

bash

```
sed -i 's/%/ percent/g' $REPORT
```

...

`-i` edits the file in place. `s/%/ percent/g` replaces every `%` symbol with the word `percent` throughout the entire report file. Since we write the report first and clean it up after, this one `sed` command handles the entire file at once.

📈 Expected Output

===== Disk Usage Report — 2024-01-15 =====

== Filesystems Above 20 percent Usage (Ranked) ==

87 percent used — /dev/sda1 mounted on / (42G of 50G)

45 percent used — /dev/sdb1 mounted on /data (90G of 200G)

23 percent used — /dev/sdc1 mounted on /backup (46G of 200G)

== Top 5 Largest Files in /var/log ==

2.1G /var/log/syslog

845M /var/log/auth.log

210M /var/log/kern.log

98M /var/log/dpkg.log

45M /var/log/apt/history.log

===== Report saved to: /tmp/disk_report.txt =====

Module 6 — Error Handling & Debugging

This is what separates amateur scripts from production-grade ones. A script that fails silently in production can cause serious damage. Proper error handling makes your scripts safe, predictable, and easy to troubleshoot.

1. Exit Codes — The Foundation of Error Handling

Every command in Linux returns an exit code. 0 means success, anything else means failure:

```
bash
```

```
#!/bin/bash
```

```
# Check exit code of last command using $?
```

```
ls /etc/passwd
```

```
echo "Exit code: $" # 0 — success
```

```
ls /nonexistent
```

```
echo "Exit code: $" # 2 — failure
```

```
# Always check exit codes for critical commands
```

```
useradd testuser
```

```
if [ $? -ne 0 ]; then
```

```
    echo "✗ Failed to create user"
```

```
    exit 1
```

```
fi
```

```
echo "✓ User created successfully"
```

2. set Options — Script Safety Switches

These go near the top of every production script:

```
bash
```

```
#!/bin/bash
```

```
set -e      # Exit immediately if any command fails
```

```
set -u      # Treat unset variables as errors
```

```
set -o pipefail # Catch failures inside pipes
```

```
set -x      # Debug mode — print every command before executing
```

```
# Combine them in one line — standard in production scripts
```

```
set -euo pipefail
```

Why each one matters:

bash

```
#!/bin/bash
```

```
set -euo pipefail
```

```
# Without set -e, this script would continue even after rm fails
```

```
rm /important/file.txt
```

```
echo "This would still run without set -e — dangerous!"
```

```
# Without set -u, typos in variable names silently use empty string
```

```
echo $USERNMAE # typo — without set -u this silently returns empty
```

```
# Without pipefail, this would show exit code 0 even though grep failed
```

```
cat /nonexistent | grep "pattern"
```

```
echo $? # would show 0 without pipefail
```

3. trap — Catching Errors and Cleaning Up

trap lets you run code automatically when a script exits, crashes, or receives a signal:

bash

```
#!/bin/bash
```

```
set -euo pipefail
```

```
TMPFILE="/tmp/myscript_$$_temp.txt"
```

```
# --- Cleanup function ---
```

```
cleanup() {
```

```
    echo "🧹 Cleaning up temporary files..."
```

```
    rm -f $TMPFILE
```

```

echo "✓ Cleanup done"
}

# --- Trap signals ---
trap cleanup EXIT      # Run cleanup on any exit
trap cleanup ERR       # Run cleanup on error
trap "echo 'Interrupted!'; cleanup; exit 1" INT TERM  # Ctrl+C or kill

```

```

# --- Main script ---
echo "Working..." > $TMPFILE
sleep 5
echo "Done!"

```

Common trap signals:

Signal	When it fires
EXIT	Any script exit — normal or error
ERR	Any command returns non-zero
INT	User presses Ctrl+C
TERM	Script receives kill signal

4. Custom Error Function

Instead of repeating error handling everywhere, build a reusable function:

```

bash
#!/bin/bash

set -euo pipefail

```

```

LOGFILE="/var/log/myscript.log"

```

```

# --- Logging functions ---
log_info() {

```

```

echo "[$(date '+%Y-%m-%d %H:%M:%S')] [INFO] $1" | tee -a $LOGFILE
}

log_warn() {
    echo "[$(date '+%Y-%m-%d %H:%M:%S')] [WARN] $1" | tee -a $LOGFILE
}

log_error() {
    echo "[$(date '+%Y-%m-%d %H:%M:%S')] [ERROR] $1" | tee -a $LOGFILE
    exit 1
}

# --- Usage ---
log_info "Script started"
log_warn "Disk usage is high"
log_error "Failed to connect to database" # This will exit the script
log_info "This line will never run"

```

5. Debugging with set -x

set -x prints every command before executing it — invaluable for troubleshooting:

bash

```
#!/bin/bash
```

```
# Enable debug for entire script
```

```
set -x
```

```
USERNAME="john"
```

```
HOME_DIR="/home/$USERNAME"
```

```
if [ -d "$HOME_DIR" ]; then
    echo "Home exists"
fi
```

```

**\*\*Output with `set -x`:\*\***

```

```
+ USERNAME=john
+ HOME_DIR=/home/john
+ '[' -d /home/john ']'
+ echo 'Home exists'
Home exists
```

Debug only a specific section:

bash

#!/bin/bash

```
echo "Normal section — no debug output"
```

```
set -x  # Start debugging
DISK=$(df / | awk 'NR==2 {print $5}')
echo "Disk: $DISK"
set +x  # Stop debugging
```

```
echo "Back to normal"
```

6. Validating Inputs Safely

bash

#!/bin/bash

```
set -euo pipefail
```

```

validate_user() {

    local USERNAME=$1

    # Check not empty
    if [ -z "$USERNAME" ]; then
        log_error "Username cannot be empty"
    fi

    # Check valid characters only
    if [[ ! "$USERNAME" =~ ^[a-zA-Z0-9_-]+$ ]]; then
        log_error "Invalid username: $USERNAME — only letters, numbers, _ and - allowed"
    fi

    # Check user doesn't already exist
    if id "$USERNAME" &>/dev/null; then
        log_error "User $USERNAME already exists"
    fi

    log_info "Username $USERNAME is valid"
}

validate_ip() {

    local IP=$1

    if [[ ! "$IP" =~ ^[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}$ ]]; then
        log_error "Invalid IP address: $IP"
    fi

    log_info "IP $IP is valid"
}

```

7. Real-World Example — Production-Grade Script with Full Error Handling

bash

```
#!/bin/bash

# =====

# Script Name: safe_backup.sh

# Description: Backs up a directory safely

#       with full error handling

# Usage   : sudo ./safe_backup.sh <source_dir>

# =====

set -euo pipefail

# --- Config ---

BACKUP_DIR="/backup"

LOGFILE="/var/log/backup.log"

TIMESTAMP=$(date +%Y%m%d_%H%M%S)

TMPFILE="/tmp/backup_$$_${TIMESTAMP}.tar.gz"

# --- Logging ---

log_info() { echo "[${date '+%Y-%m-%d %H:%M:%S'}] [INFO] $1" | tee -a $LOGFILE; }

log_warn() { echo "[${date '+%Y-%m-%d %H:%M:%S'}] [WARN] $1" | tee -a $LOGFILE; }

log_error() { echo "[${date '+%Y-%m-%d %H:%M:%S'}] [ERROR] $1" | tee -a $LOGFILE; exit 1; }

# --- Cleanup on exit ---

cleanup() {

    if [ -f "$TMPFILE" ]; then

        log_warn "Removing temporary file: $TMPFILE"

        rm -f "$TMPFILE"

    fi}
```

```

}

trap cleanup EXIT

# --- Validation ---

[ "$EUID" -ne 0 ] && log_error "Must run as root"
[ -z "${1:-}" ] && log_error "Usage: $0 <source_directory>"
[ ! -d "$1" ] && log_error "Source directory not found: $1"

SOURCE=$1

DEST="$BACKUP_DIR/${basename $SOURCE}_${TIMESTAMP}.tar.gz"

# --- Check backup directory ---

if [ ! -d "$BACKUP_DIR" ]; then
    log_info "Creating backup directory: $BACKUP_DIR"
    mkdir -p "$BACKUP_DIR" || log_error "Failed to create backup directory"
fi

# --- Disk space check ---

AVAILABLE=$(df "$BACKUP_DIR" | awk 'NR==2 {print $4}')
SOURCE_SIZE=$(du -sk "$SOURCE" | cut -f1)

if [ "$SOURCE_SIZE" -ge "$AVAILABLE" ]; then
    log_error "Not enough disk space. Need: ${SOURCE_SIZE}K Available: ${AVAILABLE}K"
fi

# --- Run backup ---

log_info "Starting backup: $SOURCE → $DEST"

tar -czf "$TMPFILE" "$SOURCE" || log_error "tar command failed"

```

```
mv "$TMPFILE" "$DEST" || log_error "Failed to move backup to destination"

log_info "✓ Backup completed successfully: $DEST"
log_info " Size: $(du -sh $DEST | cut -f1)"
```

Key Takeaways from Module 6

- Always start production scripts with set -euo pipefail
 - Use trap cleanup EXIT to always clean up temp files even on crash
 - Build log_info, log_warn, log_error functions in every script
 - Use set -x to debug — wrap only the section you need to troubleshoot
 - Always validate inputs before using them — check empty, type, and existence
 - \$? captures exit codes — always check them after critical commands
 - Never assume a command succeeded — verify it
-

Ready for **Module 7 — Real-World Sysadmin Projects** where we tie everything together into full production scripts? 

Yes

12:53 PM

Module 7 — Real-World Sysadmin Projects

This is where everything comes together. We'll build 5 complete production-grade scripts that you can actually use in your daily work as a Linux administrator.

Project 1 — Automated User Management Script

bash

```
#!/bin/bash

# =====

# Script Name: user_manager.sh

# Description: Create, delete, and list users

#       with full validation and logging
```

```

# Usage : sudo ./user_manager.sh {create|delete|list} [username]
# =====
set -euo pipefail

LOGFILE="/var/log/user_manager.log"
DEFAULT_SHELL="/bin/bash"
DEFAULT_GROUP="users"

# --- Logging ---
log_info() { echo "[$(date '+%Y-%m-%d %H:%M:%S')] [INFO] $1" | tee -a $LOGFILE; }
log_warn() { echo "[$(date '+%Y-%m-%d %H:%M:%S')] [WARN] $1" | tee -a $LOGFILE; }
log_error() { echo "[$(date '+%Y-%m-%d %H:%M:%S')] [ERROR] $1" | tee -a $LOGFILE; exit 1; }

# --- Root check ---
[ "$EUID" -ne 0 ] && log_error "Must run as root. Use sudo."

# --- Validate username ---
validate_username() {
local USERNAME=$1
[ -z "$USERNAME" ] && log_error "Username cannot be empty"
[[ ! "$USERNAME" =~ ^[a-zA-Z0-9_-]+$ ]] && \
log_error "Invalid username: only letters, numbers, _ and - allowed"
}

# --- Create user ---
create_user() {
local USERNAME=$1
validate_username "$USERNAME"

```

```

if id "$USERNAME" &>/dev/null; then
    log_warn "User $USERNAME already exists — skipping"
    return
fi

# Create user with home directory and default shell
useradd -m -s "$DEFAULT_SHELL" -g "$DEFAULT_GROUP" "$USERNAME" \
    || log_error "Failed to create user: $USERNAME"

# Set password expiry — force change on first login
chage -d 0 "$USERNAME"

# Set a temporary password
echo "$USERNAME:TempPass@123" | chpasswd \
    || log_error "Failed to set password for: $USERNAME"

log_info "✓ User created: $USERNAME | Home: /home/$USERNAME | Shell: $DEFAULT_SHELL"
log_warn "Temporary password set. User must change on first login."
}

# --- Delete user ---
delete_user() {
    local USERNAME=$1
    validate_username "$USERNAME"

    if ! id "$USERNAME" &>/dev/null; then
        log_warn "User $USERNAME does not exist — skipping"
        return
    fi
}

```

```

# Archive home directory before deletion

ARCHIVE="/backup/${USERNAME}_home_$(date +%Y%m%d).tar.gz"
mkdir -p /backup
tar -czf "$ARCHIVE" "/home/$USERNAME" 2>/dev/null \
  && log_info "Home directory archived to: $ARCHIVE"

userdel -r "$USERNAME" || log_error "Failed to delete user: $USERNAME"
log_info "✓ User deleted: $USERNAME | Archive: $ARCHIVE"
}

# --- List users ---
list_users() {
  echo ""
  echo "===== System Users with Bash Shell ====="
  printf "%-15s %-25s %-10s\n" "USERNAME" "HOME" "LAST LOGIN"
  echo "-----"
  while IFS=: read -r USER _ _ _ HOME SHELL; do
    if [ "$SHELL" = "/bin/bash" ]; then
      LAST=$(lastlog -u "$USER" 2>/dev/null | awk 'NR==2 {print $4,$5,$9}')
      printf "%-15s %-25s %-10s\n" "$USER" "$HOME" "$LAST"
    fi
  done < /etc/passwd
  echo ""
}

# --- Main ---
ACTION=${1:-}
USERNAME=${2:-}

```

```
case "$ACTION" in
    create) create_user "$USERNAME" ;;
    delete) delete_user "$USERNAME" ;;
    list) list_users ;;
    *) echo "Usage: $0 {create|delete|list} [username]"
        exit 1
    ;;
esac
```

How to run:

```
bash
sudo ./user_manager.sh create john
sudo ./user_manager.sh delete john
sudo ./user_manager.sh list
```

Project 2 — System Health Monitor

```
bash
#!/bin/bash
# =====
# Script Name: health_monitor.sh
# Description: Full system health check with
#               alerting and detailed reporting
# Usage   : sudo ./health_monitor.sh
# Schedule : */5 * * * * /path/to/health_monitor.sh
# =====
set -euo pipefail

# --- Config ---
```

```

LOGFILE="/var/log/health_monitor.log"
REPORT="/tmp/health_report_${date +%Y%m%d_%H%M%S}.txt"
DISK_THRESHOLD=85
MEM_THRESHOLD=90
CPU_THRESHOLD=80
SERVICES=("sshd" "cron" "rsyslog")
ALERT_EMAIL="admin@yourdomain.com"

# --- Logging ---
log_info() { echo "[${date '+%Y-%m-%d %H:%M:%S'}] [INFO] $1" | tee -a $LOGFILE; }
log_warn() { echo "[${date '+%Y-%m-%d %H:%M:%S'}] [WARN] $1" | tee -a $LOGFILE; }
log_error() { echo "[${date '+%Y-%m-%d %H:%M:%S'}] [ERROR] $1" | tee -a $LOGFILE; }
log_crit() { echo "[${date '+%Y-%m-%d %H:%M:%S'}] [CRIT] $1" | tee -a $LOGFILE; }

# --- Alert function ---
send_alert() {
    local SUBJECT=$1
    local MESSAGE=$2
    echo "$MESSAGE" | mail -s "$SUBJECT" "$ALERT_EMAIL" 2>/dev/null \
        && log_info "Alert sent to $ALERT_EMAIL" \
        || log_warn "Failed to send alert email"
}

# --- Check disk ---
check_disk() {
    log_info "--- Checking Disk Usage ---"
    while read -r USAGE FS MOUNT; do
        if [ "$USAGE" -ge "$DISK_THRESHOLD" ]; then
            log_crit "Disk CRITICAL: $MOUNT is at ${USAGE}% on $FS"
        fi
    done
}

```

```

send_alert "🚨 DISK CRITICAL: $MOUNT" \
"Disk usage on $MOUNT ($FS) is at ${USAGE}% — above ${DISK_THRESHOLD}% threshold"

elif [ "$USAGE" -ge 70 ]; then
    log_warn "Disk WARNING: $MOUNT is at ${USAGE}% on $FS"
else
    log_info "Disk OK: $MOUNT is at ${USAGE}%"
fi

done <<(df -h | awk 'NR>1 {gsub(/%/,"",$5); print $5, $1, $6}')
}

# --- Check memory ---
check_memory() {
    log_info "--- Checking Memory Usage ---"

    local TOTAL=$(free -m | awk '/Mem:/ {print $2}')
    local USED=$(free -m | awk '/Mem:/ {print $3}')
    local FREE=$(free -m | awk '/Mem:/ {print $4}')
    local PERCENT=$(( USED * 100 / TOTAL ))
    local SWAP_USED=$(free -m | awk '/Swap:/ {print $3}')

    if [ "$PERCENT" -ge "$MEM_THRESHOLD" ]; then
        log_crit "Memory CRITICAL: ${PERCENT}% used (${USED}MB / ${TOTAL}MB)"
        send_alert "🚨 MEMORY CRITICAL" "Memory usage is at ${PERCENT}%"
    else
        log_info "Memory OK: ${PERCENT}% used | Free: ${FREE}MB | Swap used: ${SWAP_USED}MB"
    fi
}

# --- Check CPU ---
check_cpu() {

```

```

log_info "--- Checking CPU Usage ---"

local CPU_IDLE=$(top -bn1 | grep "Cpu(s)" | awk '{print $8}' | tr -d '%')

local CPU_USED=$(echo "100 - $CPU_IDLE" | bc 2>/dev/null || echo "N/A")

local LOAD=$(uptime | awk -F'load average:' '{print $2}' | xargs)

if [ "$CPU_USED" != "N/A" ] && [ "${CPU_USED%.%}" -ge "$CPU_THRESHOLD" ]; then
    log_warn "CPU WARNING: ${CPU_USED}% used | Load: $LOAD"
else
    log_info "CPU OK: ${CPU_USED}% used | Load average: $LOAD"
fi

}

# --- Check services ---
check_services() {
    log_info "--- Checking Services ---"

    for SERVICE in "${SERVICES[@]}"; do
        STATUS=$(systemctl is-active "$SERVICE" 2>/dev/null || echo "unknown")
        if [ "$STATUS" = "active" ]; then
            log_info "Service OK: $SERVICE is running"
        else
            log_crit "Service DOWN: $SERVICE is $STATUS — attempting restart"
            systemctl restart "$SERVICE" 2>/dev/null && \
                log_info "Service $SERVICE restarted successfully" || \
                log_error "Failed to restart $SERVICE"
        fi
    done
}

# --- Check zombie processes ---

```

```

check_zombies() {
    log_info "--- Checking Zombie Processes ---"

    local ZOMBIES=$(ps aux | awk '$8=="Z" {print $0}' | wc -l)

    if [ "$ZOMBIES" -gt 0 ]; then
        log_warn "Found $ZOMBIES zombie process(es)"

    else
        log_info "No zombie processes found"
    fi
}

# --- Generate summary report ---
generate_report() {
    echo "===== Health Report — $(date) =====" > $REPORT
    echo "" >> $REPORT
    echo "Hostname : $(hostname)" >> $REPORT
    echo "Uptime   : $(uptime -p)" >> $REPORT
    echo "Kernel   : $(uname -r)" >> $REPORT
    echo "" >> $REPORT
    grep -E "\[WARN\]|\[CRIT\]|\[ERROR\]" $LOGFILE \
        | tail -20 >> $REPORT || true
    echo "" >> $REPORT
    echo "Full log: $LOGFILE" >> $REPORT
    log_info "Report saved to: $REPORT"
}

# --- Main ---
log_info "===== Health Monitor Started ====="
check_disk
check_memory

```

```
check_cpu
check_services
check_zombies
generate_report
log_info "===== Health Monitor Completed ====="
```

Project 3 — Automated Backup Script

bash

```
#!/bin/bash

# =====
# Script Name: auto_backup.sh
# Description: Automated backup with retention
#       policy and integrity verification
# Usage   : sudo ./auto_backup.sh <source>
# Schedule : 0 2 * * * /path/to/auto_backup.sh /var/www
# =====
set -euo pipefail
```

```
BACKUP_ROOT="/backup"
LOGFILE="/var/log/auto_backup.log"
RETENTION_DAYS=7
TIMESTAMP=$(date +%Y%m%d_%H%M%S)
```

```
log_info() { echo "[${date '+%Y-%m-%d %H:%M:%S'}] [INFO] $1" | tee -a $LOGFILE; }
log_error() { echo "[${date '+%Y-%m-%d %H:%M:%S'}] [ERROR] $1" | tee -a $LOGFILE; exit 1; }
```

```
cleanup() { [ -f "${TMPFILE:-}" ] && rm -f "$TMPFILE"; }
trap cleanup EXIT
```

```

# --- Validation ---

[ "$EUID" -ne 0 ] && log_error "Must run as root"
[ -z "${1:-}" ] && log_error "Usage: $0 <source_directory>"
[ ! -d "$1" ] && log_error "Source not found: $1"

SOURCE=$1

SOURCE_NAME=$(basename "$SOURCE")
DEST_DIR="$BACKUP_ROOT/$SOURCE_NAME"
DEST_FILE="$DEST_DIR/${SOURCE_NAME}_${TIMESTAMP}.tar.gz"
TMPFILE="$DEST_DIR/.tmp_${TIMESTAMP}.tar.gz"
CHECKSUM_FILE="${DEST_FILE}.md5"

mkdir -p "$DEST_DIR"

# --- Disk space check ---

AVAILABLE=$(df "$BACKUP_ROOT" | awk 'NR==2 {print $4}')
SOURCE_SIZE=$(du -sk "$SOURCE" | cut -f1)
[ "$SOURCE_SIZE" -ge "$AVAILABLE" ] && \
log_error "Insufficient disk space. Need: ${SOURCE_SIZE}K | Available: ${AVAILABLE}K"

# --- Run backup ---

log_info "Starting backup: $SOURCE → $DEST_FILE"
tar -czf "$TMPFILE" -C "$(dirname $SOURCE)" "$SOURCE_NAME" \
|| log_error "Backup failed during compression"

mv "$TMPFILE" "$DEST_FILE"

# --- Integrity check ---

md5sum "$DEST_FILE" > "$CHECKSUM_FILE"

```

```

md5sum -c "$CHECKSUM_FILE" &>/dev/null \
&& log_info "✓ Integrity check passed" \
|| log_error "✗ Integrity check FAILED — backup may be corrupt"

BACKUP_SIZE=$(du -sh "$DEST_FILE" | cut -f1)

log_info "✓ Backup complete | Size: $BACKUP_SIZE | File: $DEST_FILE"

# --- Retention policy ---

log_info "Applying retention policy: keeping last $RETENTION_DAYS days"
find "$DEST_DIR" -name "*.tar.gz" -mtime +$RETENTION_DAYS -exec rm -f {} \; \
&& log_info "Old backups cleaned up successfully"

REMAINING=$(find "$DEST_DIR" -name "*.tar.gz" | wc -l)

log_info "Total backups retained: $REMAINING"

```

Project 4 — Log Rotation & Cleanup Script

```

bash

#!/bin/bash

# =====

# Script Name: log_cleanup.sh

# Description: Rotates, compresses and cleans

#      up old logs automatically

# Usage   : sudo ./log_cleanup.sh

# Schedule : 0 0 * * 0 /path/to/log_cleanup.sh

# =====

set -euo pipefail

```

```

LOG_DIR="/var/log"
ARCHIVE_DIR="/var/log/archive"

```

```

LOGFILE="/var/log/log_cleanup.log"
COMPRESS_DAYS=3
DELETE_DAYS=30
MAX_SIZE_MB=100

log_info() { echo "[$(date '+%Y-%m-%d %H:%M:%S')] [INFO] $1" | tee -a $LOGFILE; }
log_warn() { echo "[$(date '+%Y-%m-%d %H:%M:%S')] [WARN] $1" | tee -a $LOGFILE; }

[ "$EUID" -ne 0 ] && { echo "Must run as root"; exit 1; }

mkdir -p "$ARCHIVE_DIR"

log_info "===== Log Cleanup Started ====="

# --- Compress logs older than N days ---
log_info "Compressing logs older than $COMPRESS_DAYS days..."
find "$LOG_DIR" -maxdepth 1 -type f -name "*.log" \
-mtime +$COMPRESS_DAYS ! -name "*.gz" \
| while read -r LOGF; do
    gzip -f "$LOGF" && log_info "Compressed: $LOGF"
done

# --- Move compressed logs to archive ---
log_info "Moving compressed logs to archive..."
find "$LOG_DIR" -maxdepth 1 -name "*.gz" \
| while read -r GZFILE; do
    mv "$GZFILE" "$ARCHIVE_DIR/" \
    && log_info "Archived: $(basename $GZFILE)"
done

```

```

# --- Delete archive logs older than retention period ---

log_info "Deleting archives older than $DELETE_DAYS days..."

find "$ARCHIVE_DIR" -name "*.gz" -mtime +$DELETE_DAYS \
| while read -r OLD; do
    rm -f "$OLD" && log_info "Deleted: $(basename $OLD)"
done

# --- Truncate oversized active logs ---

log_info "Checking for oversized active logs..."

find "$LOG_DIR" -maxdepth 1 -type f -name "*.\log" \
| while read -r LOGF; do
    SIZE_MB=$(du -m "$LOGF" | cut -f1)
    if [ "$SIZE_MB" -ge "$MAX_SIZE_MB" ]; then
        log_warn "$LOGF is ${SIZE_MB}MB — truncating"
        cp "$LOGF" "${LOGF}.bak"
        : > "$LOGF"
        log_info "Truncated: $LOGF | Backup: ${LOGF}.bak"
    fi
done

# --- Summary ---

ARCHIVE_SIZE=$(du -sh "$ARCHIVE_DIR" | cut -f1)
ARCHIVE_COUNT=$(find "$ARCHIVE_DIR" -name "*.gz" | wc -l)
log_info "Archive size: $ARCHIVE_SIZE | Files: $ARCHIVE_COUNT"
log_info "===== Log Cleanup Completed ====="

```

Project 5 — Server Inventory Report

bash

```

#!/bin/bash

# =====

# Script Name: server_inventory.sh

# Description: Generates a full server
#               inventory and hardware report

# Usage   : sudo ./server_inventory.sh

# =====

set -euo pipefail

REPORT="/tmp/inventory_$(hostname)_$(date +%Y%m%d).txt"

# --- Helper ---

section() { echo "" | tee -a $REPORT; echo "===== $1 =====" | tee -a $REPORT; }

info()  { printf "%-20s: %s\n" "$1" "$2" | tee -a $REPORT; }

echo "===== Server Inventory Report =====" | tee $REPORT
echo "Generated: $(date)" | tee -a $REPORT

# --- System Info ---

section "System Information"

info "Hostname"    "$(hostname -f)"

info "OS"          "$(grep PRETTY_NAME /etc/os-release | cut -d= -f2 | tr -d '\"')"

info "Kernel"      "$(uname -r)"

info "Architecture" "$(uname -m)"

info "Uptime"      "$(uptime -p)"

info "Last Boot"   "$(who -b | awk '{print $3, $4}')"

# --- CPU Info ---

section "CPU Information"

```

```
info "Model"      "$(grep 'model name' /proc/cpuinfo | head -1 | cut -d: -f2 | xargs)"
info "CPU Cores"   "$(nproc)"
info "CPU Sockets"  "$(grep 'physical id' /proc/cpuinfo | sort -u | wc -l)"
info "Load Average"  "$(uptime | awk -F'load average:' '{print $2}' | xargs)"
```

--- Memory Info ---

section "Memory Information"

```
info "Total RAM"    "$(free -h | awk '/Mem:/ {print $2})"
info "Used RAM"     "$(free -h | awk '/Mem:/ {print $3})"
info "Free RAM"      "$(free -h | awk '/Mem:/ {print $4})"
info "Total Swap"    "$(free -h | awk '/Swap:/ {print $2})"
info "Used Swap"     "$(free -h | awk '/Swap:/ {print $3})"
```

--- Disk Info ---

section "Disk Information"

```
df -h | awk 'NR>1 {printf "%-20s %-8s %-8s %-8s %s\n", $1, $2, $3, $4, $5}' | tee -a $REPORT
```

--- Network Info ---

section "Network Information"

```
ip -br addr show | tee -a $REPORT
echo "" | tee -a $REPORT
info "Default Gateway" "$(ip route | awk '/default/ {print $3})"
info "DNS Servers"     "$(grep nameserver /etc/resolv.conf | awk '{print $2}' | tr '\n' '')"
```

--- Running Services ---

section "Active Services"

```
systemctl list-units --type=service --state=active \
| awk 'NR>1 && /running/ {print $1}' \
| head -20 | tee -a $REPORT
```

```
# --- Logged in Users ---
section "Currently Logged In Users"
who | tee -a $REPORT

# --- Last 5 Logins ---
section "Last 5 Logins"
last | head -5 | tee -a $REPORT

echo "" | tee -a $REPORT
echo "===== Report saved to: $REPORT =====" | tee -a $REPORT
```

Scheduling Scripts with Cron

All these scripts become truly powerful when automated with cron:

bash

```
# Edit crontab
```

```
crontab -e
```

```
# Cron syntax:
```

```
# MIN HOUR DAY MONTH WEEKDAY COMMAND
```

```
# * * * * *
```

```
# Run health monitor every 5 minutes
```

```
*/5 * * * * /opt/scripts/health_monitor.sh
```

```
# Run backup daily at 2 AM
```

```
0 2 * * * /opt/scripts/auto_backup.sh /var/www
```

```
# Run log cleanup every Sunday at midnight
```

```
0 0 * * 0 /opt/scripts/log_cleanup.sh

# Run inventory report every Monday at 7 AM
0 7 * * 1 /opt/scripts/server_inventory.sh

# Run user audit daily at 6 AM
0 6 * * * /opt/scripts/user_audit.sh
```

Key Takeaways from Module 7

- Always start every production script with `set -euo pipefail`
- Always include logging functions — every action should be traceable
- Always validate inputs before doing anything destructive
- Use `trap cleanup EXIT` to handle unexpected failures gracefully
- Use `md5sum` to verify backup integrity — never assume a backup is good
- Use `find` with `-mtime` for age-based file management
- Schedule scripts with cron — automation only works when it runs automatically
- Always test scripts with non-critical data before running in production

Bash Scripting Course Complete!

You've now covered everything from foundations to production-grade scripts. Here's what you've mastered:

Module	Skills Gained
1 — Foundations	Script structure, variables, arguments
2 — Conditionals	<code>if/else</code> , <code>case</code> , test operators
3 — Loops	<code>for</code> , <code>while</code> , <code>until</code> , arrays
4 — Functions	Reusable code, local variables, return values
5 — File & Text	<code>grep</code> , <code>awk</code> , <code>sed</code> , <code>cut</code> , <code>sort</code> , <code>uniq</code>
6 — Error Handling	<code>set -euo pipefail</code> , <code>trap</code> , debugging

Module	Skills Gained
7 — Real Projects	5 production-ready sysadmin scripts

Ready to move on to **Python?** 

Python for Linux Administrators

Before we dive in, let me quickly assess where you're starting from so I can tailor this perfectly.

Q: What is your current Python level? A: Complete beginner

Q: What are your main goals with Python? (Select all that apply) A: Automate sysadmin tasks, Replace complex bash scripts, Write production-grade tools, Build a strong base for Ansible/Terraform

10:09 PM

Perfect! Starting from scratch with all the right goals. Since you already know Bash well, I'll constantly draw comparisons to Bash so Python feels familiar and natural rather than foreign.

Here's your complete Python learning path:

Your Python Learning Path

Module 1 — Python Foundations How Python differs from Bash, variables, data types, input/output, running scripts. Building the right mental model from day one.

Module 2 — Control Flow if/elif/else, for loops, while loops, break/continue. You already know the logic — just new syntax.

Module 3 — Functions Defining functions, arguments, return values, scope. Cleaner and more powerful than Bash functions.

Module 4 — Data Structures Lists, dictionaries, tuples, sets. This is where Python becomes dramatically more powerful than Bash.

Module 5 — Working with Files & OS Reading/writing files, os and subprocess modules. Replacing your Bash file operations with Python.

Module 6 — Error Handling try/except, logging module, writing robust scripts. The Python equivalent of set -euo pipefail.

Module 7 — Sysadmin Libraries subprocess, shutil, psutil, paramiko, argparse. The tools that make Python unstoppable for sysadmins.

Module 8 — Real-World Sysadmin Projects 5 production-grade Python scripts replacing and improving on your Bash scripts.

Each module has concept explanations with Bash comparisons, real sysadmin examples, and hands-on exercises.

Shall we dive into **Module 1 — Python Foundations** right now? 

yes

Feb 25

Module 1 — Python Foundations

The first thing to know: Python is not just "better Bash". It's a full programming language. But as a Linux admin who knows Bash, you already think like a programmer — Python just gives you cleaner tools.

1. Python vs Bash — The Mental Model

	Bash	Python
Best for	Quick system commands, pipelines	Complex logic, data processing, tools
Variables	NAME="john"	name = "john"
Running	./script.sh	python3 script.py
Output	echo "hello"	print("hello")
Comments	# comment	# comment
String + var	"Hello \$NAME"	f"Hello {name}"
Exit on error	set -e	Exceptions handle this

2. Your First Python Script

```
python  
#!/usr/bin/env python3  
# ======  
# Script Name: hello_admin.py  
# Description: First Python script  
# Usage    : python3 hello_admin.py  
# ======  
  
# Print to screen — equivalent of echo in Bash
```

```
print("Hello, Linux Admin!")
```

```
print("Welcome to Python.")
```

Save it as `hello_admin.py` and run it:

bash

```
python3 hello_admin.py
```

Notice `#!/usr/bin/env python3` instead of `#!/bin/bash` — same idea, different interpreter.

3. Variables & Data Types

In Bash everything is a string. In Python, variables have **types** — this is the biggest mindset shift:

python

```
#!/usr/bin/env python3
```

```
# String — text
```

```
hostname = "web-server-01"
```

```
username = "john"
```

```
# Integer — whole number
```

```
port    = 22
```

```
cpu_cores = 4
```

```
# Float — decimal number
```

```
disk_used = 87.5
```

```
# Boolean — True or False (capital T and F — unlike bash)
```

```
is_root  = True
```

```
is_active = False
```

```
# Print variables
```

```
print(hostname)
```

```
print(port)
print(is_root)

# Check the type of any variable
print(type(hostname)) # <class 'str'>
print(type(port))    # <class 'int'>
print(type(is_root)) # <class 'bool'>
```

4. Strings — Much More Powerful Than Bash

```
python
#!/usr/bin/env python3
```

```
hostname = "web-server-01"
ip      = "192.168.1.10"
port    = 22
```

```
# f-strings — cleanest way to embed variables (like "$VAR" in Bash)
print(f"Connecting to {hostname} at {ip}:{port}")
```

```
# String methods — built-in tools
print(hostname.upper())      # WEB-SERVER-01
print(hostname.replace("-", "_"))# web_server_01
print(hostname.startswith("web"))# True
print(hostname.split("-"))    # ['web', 'server', '01']
print(len(hostname))         # 13
```

```
# Multi-line string
message = """
Server : {hostname}
```

```
IP    : {ip}
Port  : {port}
""".format(hostname=hostname, ip=ip, port=port)
print(message)
```

```
# Check if substring exists
log_line = "ERROR: disk usage above 90%"
if "ERROR" in log_line:
    print("Error found in log!")
```

5. Getting User Input

```
python
#!/usr/bin/env python3
```

```
# Basic input — equivalent of read -p in Bash
username = input("Enter username: ")
print(f"You entered: {username}")

# Input is always a STRING — convert if you need a number
threshold = input("Enter disk threshold (number): ")
threshold = int(threshold) # convert string to integer
```

```
if threshold > 100:
    print("Threshold cannot exceed 100")
else:
    print(f"Threshold set to {threshold}%")
```

! Unlike Bash, `input()` always returns a string. You must convert to `int()` or `float()` if you need a number.

6. Command Line Arguments

In Bash you used \$1, \$2. In Python use sys.argv:

```
python  
#!/usr/bin/env python3  
import sys  
  
# sys.argv is a list of arguments  
# sys.argv[0] = script name (like $0 in Bash)  
# sys.argv[1] = first argument (like $1 in Bash)  
  
# Check argument count  
if len(sys.argv) < 2:  
    print(f"Usage: {sys.argv[0]} <username>")  
    sys.exit(1) # equivalent of exit 1 in Bash
```

```
username = sys.argv[1]  
print(f"Processing user: {username}")
```

Run it:

```
bash  
python3 script.py john  
python3 script.py      # triggers usage message
```

7. Printing & Formatting Output

```
python  
#!/usr/bin/env python3
```

```
# Basic print  
print("Hello")  
  
# Print multiple items
```

```

print("Server:", "web01", "Port:", 22)

# f-string formatting

name = "web01"

cpu = 87.5

status = "active"

print(f"Server: {name} | CPU: {cpu}% | Status: {status}")

# Formatted table output — like printf in Bash

print(f"{'USERNAME':<15} {'HOME':<25} {'STATUS':<10}")

print("-" * 50)

print(f"{'john':<15} {'/home/john':<25} {'active':<10}")

print(f"{'mary':<15} {'/home/mary':<25} {'inactive':<10}")

# The :<15 means left-align in 15 character wide column

...

```

Output:

...

USERNAME	HOME	STATUS
john	/home/john	active
mary	/home/mary	inactive

8. Indentation — Python's Most Important Rule

In Bash you use fi, done, } to close blocks. In Python **indentation IS the structure**:

python

```
#!/usr/bin/env python3
```

```
# CORRECT — consistent 4-space indentation  
disk_usage = 87
```

```
if disk_usage > 85:  
    print("Warning: disk is high") # 4 spaces  
    print("Please investigate") # 4 spaces  
else:  
    print("Disk is OK") # 4 spaces
```

```
# WRONG — will crash with IndentationError
```

```
if disk_usage > 85:  
    print("Warning") # 2 spaces  
    print("Problem") # 4 spaces — MISMATCH, will crash
```

⚠ Always use **4 spaces** for indentation. Never mix spaces and tabs. This is the most common beginner mistake.

9. Running Python Scripts

```
bash
```

```
# Run directly
```

```
python3 script.py
```

```
# Make executable and run like a bash script
```

```
chmod +x script.py
```

```
./script.py # only works if #!/usr/bin/env python3 is at top
```

```
# Check Python version
```

```
python3 --version
```

```
# Run a quick one-liner
```

```
python3 -c "print('hello from command line')"
```

```
# Interactive Python shell — great for testing  
python3  
>>> print("testing")  
>>> exit()
```

10. Python Script Template — Your Starting Point

Just like your Bash template, use this for every Python script:

```
python  
#!/usr/bin/env python3  
# ======  
# Script Name: template.py  
# Description: What this script does  
# Author   : Your Name  
# Usage    : python3 template.py <argument>  
# ======  
  
import sys  
import os  
  
# — Constants ——————  
LOGFILE  = "/var/log/myscript.log"  
THRESHOLD = 85  
  
# — Functions ——————  
def check_root():  
    """Check if script is running as root."""  
    if os.geteuid() != 0:
```

```

print("Error: This script must be run as root.")
sys.exit(1)

def validate_args():
    """Validate command line arguments."""
    if len(sys.argv) < 2:
        print(f"Usage: {sys.argv[0]} <argument>")
        sys.exit(1)
    return sys.argv[1]

# — Main ——————
def main():
    check_root()
    arg = validate_args()
    print(f"Running with argument: {arg}")

# — Entry Point ——————
if __name__ == "__main__":
    main()

```

The if `__name__ == "__main__"`: block is standard Python. It means "only run `main()` if this script is executed directly, not if it's imported by another script."

Key Takeaways from Module 1

- Use `#!/usr/bin/env python3` at the top of every script
- Variables have types — string, int, float, bool
- Use f-strings `f"Hello {name}"` for clean string formatting
- `input()` always returns a string — convert with `int()` or `float()`
- Use `sys.argv` for command line arguments, `sys.exit(1)` to exit with error
- Indentation is everything — always use 4 spaces, never mix with tabs

- if `__name__ == "__main__"`: is the standard entry point pattern
-

Your Exercise — Module 1

Write a Python script called `server_info.py` that does the same thing as your `server_info.sh` from Bash Module 1 but in Python:

1. Accepts a name as a command line argument — if none given, print usage and exit
 2. Greets the user by name
 3. Displays: hostname, current logged-in user, current date & time
 4. Shows a formatted table-style output with proper alignment
 5. As a bonus — check if the script is running as root and print that status too
-

Try it and paste your script here, or say "show solution" and we move to **Module 2 — Control Flow!** 

show solution and move to next module

10:12 PM

Solution — `server_info.py`

python

```
#!/usr/bin/env python3

# =====

# Script Name: server_info.py

# Description: Displays server information

# Author : Linux Admin

# Usage : python3 server_info.py <your_name>

# =====
```

```
import sys

import os

import getpass

from datetime import datetime
```

```
# — Constants ——————
```

```
SEPARATOR = "=" * 40
```

```
# — Functions ——————
```

```
def validate_args():  
    """Check a name argument was passed."""  
    if len(sys.argv) < 2:  
        print(f"Usage: {sys.argv[0]} <your_name>")  
        sys.exit(1)  
    return sys.argv[1]
```

```
def get_server_info():
```

```
    """Collect server information and return as dictionary."""  
    return {  
        "hostname" : os.uname().nodename,  
        "current_user": getpass.getuser(),  
        "date_time" : datetime.now().strftime("%Y-%m-%d %H:%M:%S"),  
        "running_root": "Yes" if os.geteuid() == 0 else "No",  
    }
```

```
def display_info(name, info):
```

```
    """Print formatted server info report."""  
    print(SEPARATOR)  
    print(f" Welcome, {name}!")  
    print(SEPARATOR)  
    print(f" {'Hostname':<15}: {info['hostname']}")  
    print(f" {'Logged in as':<15}: {info['current_user']}")  
    print(f" {'Date & Time':<15}: {info['date_time']}")  
    print(f" {'Running as root':<15}: {info['running_root']}")
```

```
print(SEPARATOR)

# — Main ——————
def main():

    name = validate_args()

    info = get_server_info()

    display_info(name, info)

# — Entry Point ——————
if __name__ == "__main__":
    main()
```

Explanation

Imports — Python's power comes from its libraries

```
python

import sys    # sys.argv for arguments, sys.exit() to quit
import os     # os.uname() for hostname, os.geteuid() for root check
import getpass # getpass.getuser() gets current logged-in user
from datetime import datetime # datetime.now() for current time
```

In Bash you called \$(hostname), \$(whoami) etc. In Python you import modules that give you the same information cleanly inside your script without spawning shell commands.

Returning a dictionary from a function

```
python

def get_server_info():

    return {

        "hostname" : os.uname().nodename,
        "current_user": getpass.getuser(),
        ...
```

```
}
```

Instead of setting multiple separate variables, we collect all related data into a dictionary and return it in one clean object. This is a pattern you'll use constantly in production Python scripts.

Formatted table output with :<15

```
python
```

```
print(f" {'Hostname':<15}: {info['hostname']}")
```

:<15 means left-align in a 15-character wide column — same as printf "%-15s" in Bash, but built right into the f-string.

Root check without subprocess

```
python
```

```
"running_root": "Yes" if os.geteuid() == 0 else "No"
```

```
---
```

`os.geteuid()` returns the effective user ID directly from Python — no need to call `whoami` or `id`. The `"Yes" if condition else "No"` is Python's inline if/else called a **ternary expression**.

```
---
```

```
## 🎯 Expected Output
```

```
---
```

```
=====
```

```
Welcome, John!
```

```
=====
```

```
Hostname    : web-server-01
```

```
Logged in as : john
```

```
Date & Time : 2024-01-15 10:35:22
```

```
Running as root: No
```

```
=====
```

Module 2 — Control Flow

You already know the logic from Bash — conditionals and loops. Python's syntax is cleaner, more readable, and significantly more powerful. Let's go through it all.

1. if / elif / else

python

```
#!/usr/bin/env python3
```

```
disk_usage = 87
```

```
# Basic if/elif/else — no brackets needed, just colon + indentation
```

```
if disk_usage >= 90:  
    print("CRITICAL: Disk usage is dangerously high!")  
  
elif disk_usage >= 80:  
    print("WARNING: Disk usage is high")  
  
elif disk_usage >= 60:  
    print("NOTICE: Disk usage is moderate")  
  
else:  
    print("OK: Disk usage is within safe limits")
```

Bash vs Python comparison:

bash

```
# Bash  
if [ $DISK -ge 90 ]; then  
    echo "CRITICAL"  
fi
```

python

```
# Python — cleaner, no brackets, no fi  
if disk_usage >= 90:  
    print("CRITICAL")
```

2. Comparison Operators

```
python

# Numbers

disk == 90 # Equal

disk != 90 # Not equal

disk > 90 # Greater than

disk < 90 # Less than

disk >= 90 # Greater than or equal

disk <= 90 # Less than or equal

# Strings

status == "active" # Equal

status != "active" # Not equal

"ERROR" in log_line # Contains substring (no grep needed!)

"ERROR" not in log_line # Does not contain

# Boolean / None checks

is_root == True # works but...

if is_root: # ...this is the Pythonic way

if not is_root: # equivalent of [ ! ]

# Check if variable is empty / None

if username: # True if not empty, not None, not 0

if not username: # True if empty or None

if username is None: # Explicit None check
```

3. Logical Operators — and, or, not

```
python
```

```

# Bash used && and ||
# Python uses plain English: and, or, not

disk_usage = 87
memory_usage = 92
service = "active"

# AND — both must be true
if disk_usage > 85 and memory_usage > 90:
    print("CRITICAL: Both disk and memory are high!")

# OR — at least one must be true
if disk_usage > 90 or memory_usage > 90:
    print("WARNING: A resource is critically high")

# NOT
if not service == "active":
    print("Service is not running")

# Combining
if disk_usage > 85 and memory_usage > 85 and service == "active":
    print("Resources high but service still running")

```

4. for Loops

```

python
#!/usr/bin/env python3

```

```

# Loop over a list — like for SERVICE in nginx sshd cron in Bash
services = ["nginx", "sshd", "cron", "rsyslog"]

```

```
for service in services:  
    print(f"Checking: {service}")  
  
# Loop with index using enumerate()  
for index, service in enumerate(services):  
    print(f"{index + 1}. {service}")  
  
# Loop over a range of numbers  
for i in range(5):      # 0, 1, 2, 3, 4  
    print(f"Attempt {i+1}")  
  
for i in range(1, 6):    # 1, 2, 3, 4, 5  
    print(f"Attempt {i}")  
  
for i in range(10, 0, -1): # countdown 10 to 1  
    print(f"Countdown: {i}")  
  
# Loop over a dictionary  
server_info = {  
    "hostname": "web01",  
    "ip"      : "192.168.1.10",  
    "port"    : 22  
}  
  
for key, value in server_info.items():  
    print(f"{key}: {value}")
```

5. while Loops

```
python
#!/usr/bin/env python3
import time

# Basic while — same concept as Bash
count = 0
max_retries = 3

while count < max_retries:
    print(f"Attempt {count + 1} of {max_retries}")
    count += 1 # Python's equivalent of COUNT=$((COUNT + 1))
    time.sleep(1)

# While with break — retry until success
import subprocess

retries = 0
max_retries = 5

while retries < max_retries:
    result = subprocess.run(["ping", "-c", "1", "8.8.8.8"],
                           capture_output=True)
    if result.returncode == 0:
        print("Network is reachable!")
        break
    else:
        retries += 1
        print(f"Network unreachable. Retry {retries}/{max_retries}...")
        time.sleep(5)
```

```
else:  
    # The else on a while runs if loop completed WITHOUT a break  
    print("CRITICAL: Network unreachable after all retries!")
```

💡 Python's while/else is unique — the else block runs only if the loop finished normally without hitting a break. Perfect for retry logic.

6. break, continue, pass

python

```
#!/usr/bin/env python3
```

```
services = ["nginx", "sshd", "root", "cron", "skip_me", "rsyslog"]
```

```
for service in services:
```

```
    # skip — like continue in Bash
```

```
    if service == "skip_me":
```

```
        print(f"Skipping {service}...")
```

```
        continue
```

```
    # stop loop — like break in Bash
```

```
    if service == "root":
```

```
        print("Reached root entry. Stopping.")
```

```
        break
```

```
    # pass — do nothing placeholder (useful during development)
```

```
    if service == "cron":
```

```
        pass # TODO: add cron check later
```

```
        print(f"Processing: {service}")
```

7. List Comprehensions — Python's Superpower

This is something Bash simply can't do cleanly. It lets you build a filtered list in one line:

```
python
```

```
#!/usr/bin/env python3
```

```
# Traditional loop to filter
```

```
services = ["nginx", "sshd", "cron", "docker", "rsyslog"]
```

```
# Find all services with more than 4 characters
```

```
long_services = []
```

```
for s in services:
```

```
    if len(s) > 4:
```

```
        long_services.append(s)
```

```
# List comprehension — same thing in ONE line
```

```
long_services = [s for s in services if len(s) > 4]
```

```
print(long_services) # ['nginx', 'rsyslog', 'docker']
```

```
# Convert all to uppercase
```

```
upper_services = [s.upper() for s in services]
```

```
print(upper_services) # ['NGINX', 'SSHD', 'CRON', 'DOCKER', 'RSYSLOG']
```

```
# Real sysadmin use — filter log lines containing ERROR
```

```
log_lines = [
```

```
    "INFO: Service started",
```

```
    "ERROR: Disk full",
```

```
    "INFO: Backup complete",
```

```
    "ERROR: Connection refused",
```

```
"WARNING: High memory"
]

errors = [line for line in log_lines if "ERROR" in line]
print(errors)
# ['ERROR: Disk full', 'ERROR: Connection refused']
```

8. Real-World Example — Service Status Checker

```
python
#!/usr/bin/env python3
# =====
# Script Name: service_checker.py
# Description: Check status of multiple services
# Usage    : python3 service_checker.py
# =====

import subprocess
import sys
from datetime import datetime

SERVICES = ["sshd", "cron", "rsyslog", "nginx", "firewalld"]

def check_service(service):
    """Check if a systemd service is active. Returns True/False."""
    result = subprocess.run(
        ["systemctl", "is-active", service],
        capture_output=True,
        text=True
    )
```

```

return result.stdout.strip() == "active"

def restart_service(service):
    """Attempt to restart a service. Returns True if successful."""
    result = subprocess.run(
        ["systemctl", "restart", service],
        capture_output=True
    )
    return result.returncode == 0

def main():
    print(f"\n{'='*50}")

    print(f" Service Status Report — {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}")
    print(f"{'='*50}")

    print(f" {'SERVICE':<15} {'STATUS':<12} {'ACTION'}")
    print(f"{'-'*45}")

    results = {"ok": 0, "restarted": 0, "failed": 0}

    for service in SERVICES:
        if check_service(service):
            status = "RUNNING"
            action = "None"
            results["ok"] += 1
        else:
            status = "DOWN"
            if restart_service(service):
                action = "Restarted OK"
                results["restarted"] += 1

```

```

else:
    action = "RESTART FAILED"
    results["failed"] += 1

print(f" {service:<15} {status:<12} {action}")

print(f"\n Summary: {results['ok']} OK | "
      f"{results['restarted']} Restarted | "
      f"{results['failed']} Failed")
print(f"{'='*50}\n")

# Exit with error code if any service failed
if results["failed"] > 0:
    sys.exit(1)

if __name__ == "__main__":
    main()
...

```

****Expected Output:****

...

=====

Service Status Report — 2024-01-15 10:35:22

=====

SERVICE	STATUS	ACTION
---------	--------	--------

sshd	RUNNING	None
cron	RUNNING	None
rsyslog	RUNNING	None

```
nginx      DOWN     Restarted OK
firewalld   DOWN     RESTART FAILED
```

Summary: 3 OK | 1 Restarted | 1 Failed

Key Takeaways from Module 2

- No brackets in if statements — just if condition: with a colon
- Use and, or, not instead of &&, ||, !
- Use in to check if something exists in a string or list — replaces grep
- for item in list: replaces for ITEM in "\${ARRAY[@]}" in Bash
- count += 1 replaces COUNT=\$((COUNT + 1))
- while/else is unique to Python — else runs only if no break occurred
- List comprehensions [x for x in list if condition] are powerful one-liners
- subprocess.run() is how Python runs system commands — replaces \$() in Bash

Your Exercise — Module 2

Write a Python script called disk_monitor.py that:

1. Has a list of mount points to check: /, /home, /var, /tmp
2. For each mount point uses subprocess to run df and parse the usage percentage
3. If usage is above 85% print CRITICAL, above 70% print WARNING, otherwise print OK
4. Uses a for loop with proper formatted output
5. At the end prints a summary of how many are OK, WARNING, and CRITICAL
6. Exits with code 1 if any mount point is CRITICAL

Say "show solution" and we move straight to **Module 3 — Functions!** 

show solution and move to next module

10:26 PM

 Solution — disk_monitor.py

```
python

#!/usr/bin/env python3

# =====

# Script Name: disk_monitor.py

# Description: Monitor disk usage across

#     multiple mount points

# Usage   : python3 disk_monitor.py

# =====

import subprocess

import sys

from datetime import datetime

# — Constants ——————  
MOUNT_POINTS  = ["/", "/home", "/var", "/tmp"]
CRITICAL_THRESHOLD = 85
WARNING_THRESHOLD = 70

# — Functions ——————  
def get_disk_usage(mount_point):
    """Run df on a mount point and return usage % as integer."""
    result = subprocess.run(
        ["df", mount_point],
        capture_output=True,
        text=True
    )
    if result.returncode != 0:
        return None
    output = result.stdout.decode("utf-8")
    lines = output.split("\n")
    for line in lines:
        if mount_point in line:
            usage_percent = int(line.split()[4])
            return usage_percent
    return None
```

```

# Parse the second line, fifth column — same as awk 'NR==2 {print $5}'

lines = result.stdout.strip().split("\n")

fields = lines[1].split()

usage = int(fields[4].replace("%", ""))

return usage


def get_status(usage):

    """Return status string based on usage percentage."""

    if usage >= CRITICAL_THRESHOLD:

        return "CRITICAL"

    elif usage >= WARNING_THRESHOLD:

        return "WARNING"

    else:

        return "OK"


def print_report(results):

    """Print formatted disk usage report."""

    print(f"\n{'='*55}")

    print(f" Disk Usage Report — {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}\n")

    print(f"{'='*55}")

    print(f" {'MOUNT POINT':<15} {'USAGE':>8} {'STATUS'}\n")

    print(f"{'-'*45}")

    for mount, usage, status in results:

        if usage is None:

            print(f" {mount:<15} {'N/A':>8} UNAVAILABLE")

        else:

            print(f" {mount:<15} {str(usage)+'%':>8} {status}")

```

```
print(f'{"*55}')
```

```
def print_summary(results):
    """Print summary counts."""
    ok     = sum(1 for _, _, s in results if s == "OK")
    warning = sum(1 for _, _, s in results if s == "WARNING")
    critical = sum(1 for _, _, s in results if s == "CRITICAL")

    print(f"\n Summary: {ok} OK | {warning} WARNING | {critical} CRITICAL\n")
    return critical
```

```
# — Main ——————
```

```
def main():
    results = []

    for mount in MOUNT_POINTS:
        usage = get_disk_usage(mount)
        status = get_status(usage) if usage is not None else "UNAVAILABLE"
        results.append((mount, usage, status))

    print_report(results)
    critical_count = print_summary(results)

    if critical_count > 0:
        sys.exit(1)
```

```
# — Entry Point ——————
```

```
if __name__ == "__main__":
```

```
main()
```

Explanation

Parsing df output in Python

```
python
```

```
lines = result.stdout.strip().split("\n")
```

```
fields = lines[1].split()
```

```
usage = int(fields[4].replace("%", ""))
```

result.stdout gives us the raw text output of df as a string. .split("\n") breaks it into a list of lines — equivalent of reading line by line. lines[1] is the second line (index 1), .split() breaks it on whitespace into columns, and fields[4] is the 5th column — the usage percentage. We strip the % and convert to int for numeric comparison. This replaces awk 'NR==2 {print \$5}' | tr -d '%' from Bash.

Counting with sum() and a generator

```
python
```

```
ok = sum(1 for _, _, s in results if s == "OK")
```

This is a compact Python pattern. For every item in results where the status equals "OK", it counts 1 and sums them all. The `_` is a Python convention meaning "I don't care about this value". Much cleaner than a manual counter variable.

Returning multiple values as a tuple

```
python
```

```
results.append((mount, usage, status))
```

```
---
```

Python functions and lists can hold tuples — groups of values packed together. We build a list of these tuples and unpack them later with `for mount, usage, status in results`.

```
---
```

```
## 📈 Expected Output
```

```
---
```

```
=====
Disk Usage Report — 2024-01-15 10:35:22
=====
```

MOUNT POINT	USAGE	STATUS
-------------	-------	--------

/	87%	CRITICAL
---	-----	----------

/home	45%	OK
-------	-----	----

/var	72%	WARNING
------	-----	---------

/tmp	5%	OK
------	----	----

Summary: 2 OK | 1 WARNING | 1 CRITICAL

Module 3 — Functions

You already used functions in Bash. Python functions are far more powerful — they can accept any type of data, return multiple values, have default arguments, and be organised into reusable modules.

1. Defining & Calling Functions

```
python
```

```
#!/usr/bin/env python3
```

```
# Basic function — same idea as Bash but cleaner
```

```
def greet_admin(name):
```

```
    print(f"Hello, {name}!")
```

```
# Call it
```

```
greet_admin("John")
```

```
greet_admin("Mary")
```

```
# Function with a return value

def get_hostname():

    import os

    return os.uname().nodename


hostname = get_hostname()

print(f"Hostname: {hostname}")
```

In Bash, functions returned text via echo. In Python, functions return values directly with return — no need to capture with \$().

2. Parameters & Default Values

python

```
#!/usr/bin/env python3
```

```
# Function with default parameter values

def check_disk(mount_point="/", threshold=85):

    import subprocess

    result = subprocess.run(["df", mount_point],
                           capture_output=True, text=True)

    lines = result.stdout.strip().split("\n")

    usage = int(lines[1].split()[4].replace("%", ""))

    if usage >= threshold:

        return False, usage # failed, usage %

    return True, usage # passed, usage %


# Call with defaults

ok, usage = check_disk()

print(f"/ — {usage}% — {'OK' if ok else 'WARNING'}")
```

```
# Call with custom values

ok, usage = check_disk("/var", threshold=70)
print(f"/var — {usage}% — {'OK' if ok else 'WARNING'}")

# Keyword arguments — order doesn't matter

ok, usage = check_disk(threshold=70, mount_point="/home")
```

3. Returning Multiple Values

In Bash you could only return a number (0-255). Python functions can return anything — including multiple values:

```
python

#!/usr/bin/env python3

import os

import getpass

from datetime import datetime
```

```
def get_system_info():

    """Collect and return multiple system values at once."""

    hostname = os.uname().nodename
    current_user = getpass.getuser()
    timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
    is_root = os.geteuid() == 0

    # Return multiple values as a tuple
    return hostname, current_user, timestamp, is_root

# Unpack the returned values

hostname, user, timestamp, is_root = get_system_info()
```

```

print(f"Host: {hostname} | User: {user} | Root: {is_root}")

# Or return a dictionary for named access

def get_system_info_dict():

    return {

        "hostname" : os.uname().nodename,
        "current_user": getpass.getuser(),
        "timestamp" : datetime.now().strftime("%Y-%m-%d %H:%M:%S"),
        "is_root" : os.geteuid() == 0
    }

info = get_system_info_dict()

print(f"Host: {info['hostname']}")

print(f"User: {info['current_user']}")

```

4. Docstrings — Documenting Your Functions

python

```
#!/usr/bin/env python3
```

```
def check_service(service_name, restart_if_down=False):
```

```
    """
```

Check if a systemd service is running.

Args:

service_name (str): Name of the service to check

restart_if_down (bool): Whether to restart if service is down

Returns:

bool: True if service is running, False otherwise

```
"""

import subprocess

result = subprocess.run(
    ["systemctl", "is-active", service_name],
    capture_output=True, text=True
)
is_active = result.stdout.strip() == "active"

if not is_active and restart_if_down:
    print(f"Restarting {service_name}...")
    subprocess.run(["systemctl", "restart", service_name])

return is_active

# Access the docstring
print(check_service.__doc__)

# Use the function
if check_service("sshd"):
    print("sshd is running")

# Use with restart option
check_service("nginx", restart_if_down=True)
```

5. The Logger Function — Python Style

In Bash you built a log() function manually. Python has a built-in logging module that is far more powerful:

```
python
```

```
#!/usr/bin/env python3

import logging
import sys


def setup_logger(logfile="/var/log/myscript.log"):
    """Set up logger to write to both console and file."""
    logger = logging.getLogger("sysadmin")
    logger.setLevel(logging.DEBUG)

    # Format — same style as our Bash logger
    formatter = logging.Formatter(
        "[%(asctime)s] [%(levelname)s] %(message)s",
        datefmt="%Y-%m-%d %H:%M:%S"
    )

    # Console handler — print to screen
    console_handler = logging.StreamHandler(sys.stdout)
    console_handler.setFormatter(formatter)

    # File handler — write to log file
    file_handler = logging.FileHandler(logfile)
    file_handler.setFormatter(formatter)

    logger.addHandler(console_handler)
    logger.addHandler(file_handler)

    return logger

# Create logger
```

```
log = setup_logger()

# Use it — same levels as our Bash logger
log.info("Script started")
log.warning("Disk usage is high")
log.error("Service failed to start")
log.critical("System is unresponsive")
log.debug("Variable value: 42") # only shows if level=DEBUG
...
```

****Output:****

```
...
[2024-01-15 10:30:01] [INFO] Script started
[2024-01-15 10:30:01] [WARNING] Disk usage is high
[2024-01-15 10:30:02] [ERROR] Service failed to start
[2024-01-15 10:30:03] [CRITICAL] System is unresponsive
```

This replaces your entire `log_info()`, `log_warn()`, `log_error()` Bash functions with one professional setup.

6. *args and **kwargs — Flexible Functions

```
python
#!/usr/bin/env python3
```

```
# *args — accept any number of positional arguments
def check_services(*services):
    """Check any number of services passed as arguments."""
    for service in services:
        print(f"Checking {service}...")

check_services("nginx")
```

```
check_services("nginx", "sshd", "cron", "rsyslog")

# **kwargs — accept any number of keyword arguments
def create_user(**user_info):
    """Create a user from keyword arguments."""
    print(f"Creating user with:")
    for key, value in user_info.items():
        print(f" {key}: {value}")

create_user(username="john", shell="/bin/bash",
           home="/home/john", group="users")
```

7. Functions Calling Functions — Clean Script Architecture

This is the pattern that makes production Python scripts easy to read and maintain:

```
python
#!/usr/bin/env python3
# =====
# Script Name: system_audit.py
# Description: Full system audit using clean
#     modular functions
# Usage   : sudo python3 system_audit.py
# =====

import os
import sys
import subprocess
import logging
from datetime import datetime
```

```
# — Setup ——————  
  
logging.basicConfig(  
    level=logging.INFO,  
    format="[%(asctime)s] [%(levelname)s] %(message)s",  
    datefmt="%Y-%m-%d %H:%M:%S",  
    handlers=[  
        logging.StreamHandler(sys.stdout),  
        logging.FileHandler("/var/log/system_audit.log")  
    ]  
)  
  
log = logging.getLogger()
```

```
SERVICES    = ["sshd", "cron", "rsyslog"]  
DISK_THRESHOLD = 85
```

```
# — Helper Functions ——————
```

```
def check_root():  
    """Ensure script runs as root."""  
    if os.geteuid() != 0:  
        log.error("Must run as root. Use sudo.")  
        sys.exit(1)  
  
def get_disk_usage(mount="/"):  
    """Return disk usage % for a mount point."""  
    result = subprocess.run(["df", mount],  
                           capture_output=True, text=True)  
    return int(result.stdout.split("\n")[1].split()[4].replace("%", ""))  
  
def is_service_running(service):
```

```
"""Return True if service is active."""

result = subprocess.run(["systemctl", "is-active", service],
                      capture_output=True, text=True)

return result.stdout.strip() == "active"
```

```
def get_memory_usage():

    """Return memory usage as percentage."""

    result = subprocess.run(["free", "-m"],
                           capture_output=True, text=True)

    lines = result.stdout.strip().split("\n")

    fields = lines[1].split()

    total, used = int(fields[1]), int(fields[2])

    return round((used / total) * 100, 1)
```

— Audit Functions ——————

```
def audit_disk():

    """Audit disk usage and log results."""

    usage = get_disk_usage("/")

    if usage >= DISK_THRESHOLD:

        log.warning(f"Disk usage at {usage}% — above {DISK_THRESHOLD}% threshold")

    else:

        log.info(f"Disk usage at {usage}% — OK")

    return usage
```

```
def audit_memory():

    """Audit memory usage and log results."""

    usage = get_memory_usage()

    if usage >= 90:

        log.warning(f"Memory usage at {usage}% — critical")
```

```

else:
    log.info(f"Memory usage at {usage}% — OK")

return usage

def audit_services():
    """Audit all services and log results."""
    results = {}

    for service in SERVICES:
        running = is_service_running(service)
        results[service] = running

        if running:
            log.info(f"Service {service} — RUNNING")
        else:
            log.error(f"Service {service} — DOWN")

    return results

def print_summary(disk, memory, services):
    """Print final summary report."""

    down_services = [s for s, ok in services.items() if not ok]

    print(f"\n{'='*45}")

    print(f" AUDIT SUMMARY — {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}")
    print(f"{'='*45}")

    print(f" Disk Usage : {disk}%")
    print(f" Memory Usage : {memory}%")
    print(f" Services OK : {sum(services.values())}/{len(services)}")

    if down_services:
        print(f" Services DOWN: {'.'.join(down_services)}")

    print(f"{'='*45}\n")

```

```
# — Main ——————  
  
def main():  
    check_root()  
    log.info("===== System Audit Started =====")  
  
    disk = audit_disk()  
    memory = audit_memory()  
    services = audit_services()  
  
    print_summary(disk, memory, services)  
  
    log.info("===== System Audit Completed =====")  
  
    # Exit with error if critical issues found  
    if disk >= DISK_THRESHOLD or not all(services.values()):  
        sys.exit(1)  
  
# — Entry Point ——————  
  
if __name__ == "__main__":  
    main()
```

Key Takeaways from Module 3

- Functions are defined with `def function_name(params):` and called by name
- Default parameter values make functions flexible: `def check(mount="/", threshold=85)`
- Functions can return multiple values as tuples — unpack with `a, b = function()`
- Return dictionaries for named access to multiple related values
- Always write docstrings "'''description''' under every function definition
- Use Python's logging module instead of manual log functions — it's far more powerful
- Use `*args` for variable positional arguments and `**kwargs` for keyword arguments

- Keep functions small and focused — one function, one job
-

Your Exercise — Module 3

Write a Python script called user_manager.py that:

1. Has a setup_logger() function that logs to screen and to /var/log/user_manager.log
 2. Has a check_root() function
 3. Has a user_exists(username) function that returns True/False
 4. Has a create_user(username, shell="/bin/bash") function that runs useradd
 5. Has a delete_user(username) function that runs userdel -r
 6. Has a list_users() function that reads /etc/passwd and prints all users with /bin/bash shell
 7. Has a clean main() that accepts create, delete, or list as a command line argument
-

Say "show solution" and we move straight to **Module 4 — Data Structures!** 

show solution and move to next module

10:34 PM

Solution — user_manager.py

python

```
#!/usr/bin/env python3
=====
# Script Name: user_manager.py
# Description: Create, delete, and list users
# Usage    : sudo python3 user_manager.py {create|delete|list} [username]
=====
```

```
import os
import sys
import subprocess
import logging
```

```
# — Logger Setup ——————  
  
def setup_logger():  
    """Set up logger to write to console and log file."""  
  
    logger = logging.getLogger("user_manager")  
  
    logger.setLevel(logging.DEBUG)  
  
  
    formatter = logging.Formatter(  
        "[%(asctime)s] [%(levelname)s] %(message)s",  
        datefmt="%Y-%m-%d %H:%M:%S"  
    )  
  
  
    console_handler = logging.StreamHandler(sys.stdout)  
    console_handler.setFormatter(formatter)  
  
  
    file_handler = logging.FileHandler("/var/log/user_manager.log")  
    file_handler.setFormatter(formatter)  
  
  
    logger.addHandler(console_handler)  
    logger.addHandler(file_handler)  
  
    return logger  
  
  
log = setup_logger()
```

```
# — Functions ——————
```

```
def check_root():  
    """Ensure script is running as root."""  
  
    if os.geteuid() != 0:  
        log.error("Must run as root. Use sudo.")  
        sys.exit(1)
```

```
def user_exists(username):
    """Return True if user already exists on the system."""
    result = subprocess.run(
        ["id", username],
        capture_output=True
    )
    return result.returncode == 0

def create_user(username, shell="/bin/bash"):
    """
    Create a new system user.

    Args:
        username (str): Username to create
        shell (str): Login shell — defaults to /bin/bash
    """
    if user_exists(username):
        log.warning(f"User '{username}' already exists — skipping")
        return False

    result = subprocess.run(
        ["useradd", "-m", "-s", shell, username],
        capture_output=True, text=True
    )

    if result.returncode == 0:
        log.info(f"User '{username}' created successfully | Shell: {shell}")
        return True
```

```
    else:  
        log.error(f"Failed to create user '{username}': {result.stderr.strip()}")  
        return False
```

```
def delete_user(username):
```

```
    """
```

```
    Delete a system user and their home directory.
```

```
Args:
```

```
    username (str): Username to delete
```

```
    """
```

```
if not user_exists(username):
```

```
    log.warning(f"User '{username}' does not exist — skipping")
```

```
    return False
```

```
result = subprocess.run(
```

```
    ["userdel", "-r", username],
```

```
    capture_output=True, text=True
```

```
)
```

```
if result.returncode == 0:
```

```
    log.info(f"User '{username}' deleted successfully")
```

```
    return True
```

```
else:
```

```
    log.error(f"Failed to delete user '{username}': {result.stderr.strip()}")
```

```
    return False
```

```
def list_users():
```

```
    """Read /etc/passwd and print all users with /bin/bash shell."""
```

```
print(f"\n{'='*50}")

print(f" {'USERNAME':<15} {'HOME DIRECTORY':<25} {'SHELL'}")
print(f" {'-'*45}")

count = 0

with open("/etc/passwd", "r") as f:
    for line in f:
        fields = line.strip().split(":")
        username, home, shell = fields[0], fields[5], fields[6]

        if shell == "/bin/bash":
            print(f" {username:<15} {home:<25} {shell}")
            count += 1

print(f" {'-'*45}")
print(f" Total bash users: {count}")
print(f"{'='*50}\n")

# — Main ————————
def main():
    check_root()

    if len(sys.argv) < 2:
        print(f"Usage: {sys.argv[0]} {{create|delete|list}} [username]")
        sys.exit(1)

    action  = sys.argv[1].lower()
    username = sys.argv[2] if len(sys.argv) > 2 else None
```

```
if action == "create":  
    if not username:  
        log.error("Username required for create action")  
        sys.exit(1)  
    create_user(username)  
  
elif action == "delete":  
    if not username:  
        log.error("Username required for delete action")  
        sys.exit(1)  
    delete_user(username)  
  
elif action == "list":  
    list_users()  
  
else:  
    log.error(f"Unknown action '{action}'. Use: create | delete | list")  
    sys.exit(1)  
  
# — Entry Point ——————  
if __name__ == "__main__":  
    main()
```

Explanation

Reading /etc/passwd the Python way

python

```
with open("/etc/passwd", "r") as f:
```

```
    for line in f:
```

```
        fields = line.strip().split(":")
```

```
username, home, shell = fields[0], fields[5], fields[6]

with open() is Python's safe file reader — it automatically closes the file even if an error occurs. .strip() removes the newline, .split(":") splits on colon just like IFS=: in Bash, and we unpack the fields we need by index.
```

Capturing subprocess errors

```
python

result = subprocess.run(["useradd", "-m", "-s", shell, username],
                       capture_output=True, text=True)

if result.returncode == 0:
    log.info(f"Created successfully")
else:
    log.error(f"Failed: {result.stderr.strip()}")
```

`capture_output=True` captures both `stdout` and `stderr`. `text=True` returns them as strings instead of bytes. `result.stderr.strip()` gives us the actual error message from the command — far better than just knowing it failed.

with open() — always use it for files

```
python

with open("/etc/passwd", "r") as f:
    ...
```

The ``with`` statement is a **context manager**. It guarantees the file is closed properly even if an exception occurs mid-read. Always use ``with open()`` instead of manually calling ``f.open()`` and ``f.close()``.

```
## 📁 Expected Output
```

```
sudo python3 user_manager.py create john
```

```
[2024-01-15 10:30:01] [INFO] User 'john' created | Shell: /bin/bash
```

```
sudo python3 user_manager.py list
```

```
=====
USERNAME   HOME DIRECTORY   SHELL
-----
root      /root           /bin/bash
john      /home/john        /bin/bash
mary      /home/mary        /bin/bash
-----
Total bash users: 3
```

```
sudo python3 user_manager.py delete john
```

```
[2024-01-15 10:30:05] [INFO] User 'john' deleted successfully
```

Module 4 — Data Structures

This is where Python becomes dramatically more powerful than Bash. In Bash you had basic arrays. Python gives you four powerful built-in data structures that make complex sysadmin tasks simple.

1. Lists — Ordered Collections

The Python equivalent of Bash arrays — but far more capable:

```
python
```

```
#!/usr/bin/env python3
```

```
# Creating lists
```

```
services = ["nginx", "sshd", "cron", "rsyslog"]
ports   = [22, 80, 443, 3306, 5432]
mixed   = ["web01", 22, True, 87.5] # can mix types
```

```
# Accessing items — zero-indexed like Bash arrays
print(services[0])  # nginx (first item)
print(services[-1]) # rsyslog (last item)
print(services[1:3]) # ['sshd', 'cron'] (slice)

# Modifying lists
services.append("docker")      # add to end
services.insert(0, "firewalld")  # insert at position
services.remove("cron")        # remove by value
popped = services.pop()        # remove and return last item
services[0] = "haproxy"         # update by index

# Useful list operations
print(len(services))          # count items
print("sshd" in services)      # True/False — replaces grep
print(sorted(services))        # sorted copy
services.sort()                # sort in place
services.reverse()              # reverse in place

# Loop over list
for service in services:
    print(f"Service: {service}")

# List of lists — like a 2D table
servers = [
    ["web01", "192.168.1.10", "active"],
    ["web02", "192.168.1.11", "active"],
    ["db01", "192.168.1.20", "inactive"],
]
```

```
for server in servers:  
    name, ip, status = server # unpack each row  
    print(f"{name:<10} {ip:<16} {status}")
```

2. Dictionaries — Key-Value Storage

The most useful data structure for sysadmins. Like a config file in memory:

```
python
```

```
#!/usr/bin/env python3
```

```
# Creating a dictionary
```

```
server = {  
    "hostname" : "web-server-01",  
    "ip"      : "192.168.1.10",  
    "port"    : 22,  
    "os"      : "RHEL 9",  
    "is_active": True  
}
```

```
# Accessing values
```

```
print(server["hostname"])      # web-server-01  
print(server.get("ip"))        # 192.168.1.10  
print(server.get("cpu", "unknown")) # "unknown" — safe default if key missing
```

```
# Modifying dictionaries
```

```
server["port"] = 2222      # update existing key  
server["memory"] = "16GB"    # add new key  
del server["is_active"]     # delete a key
```

```

# Check if key exists

if "hostname" in server:
    print("hostname key exists")

# Loop over dictionary

for key, value in server.items():
    print(f"{key}<12>: {value}")

# Just keys or just values

print(list(server.keys()))
print(list(server.values()))

# Dictionary of dictionaries — server inventory

inventory = {

    "web01": {"ip": "192.168.1.10", "role": "web",      "status": "active"},

    "web02": {"ip": "192.168.1.11", "role": "web",      "status": "active"},

    "db01": {"ip": "192.168.1.20", "role": "database", "status": "inactive"},

}

for hostname, info in inventory.items():

    print(f"{hostname}<8> | {info['ip']}<16> | {info['role']}<10> | {info['status']}")
```

3. Tuples — Immutable Pairs

Tuples are like lists but they **cannot be changed** after creation. Use them for data that should never change:

```

python

#!/usr/bin/env python3
```

```
# Creating tuples
```

```

server_record = ("web01", "192.168.1.10", 22)
dbcreds      = ("db_user", "db_password", 5432)

# Accessing — same as list
print(server_record[0]) # web01
print(server_record[-1]) # 22

# Unpacking — very common Python pattern
hostname, ip, port = server_record
print(f"Connecting to {hostname} at {ip}:{port}")

# Tuples in lists — rows of data
connections = [
    ("web01", "192.168.1.10", 22, "active"),
    ("web02", "192.168.1.11", 22, "active"),
    ("db01", "192.168.1.20", 5432, "inactive"),
]
print(f"\n{'HOST':<10} {'IP':<16} {'PORT':<8} {'STATUS'}")
print("-" * 45)
for host, ip, port, status in connections:
    print(f"{host:<10} {ip:<16} {str(port):<8} {status}")

```

4. Sets — Unique Collections

Sets automatically remove duplicates. Perfect for finding unique values in logs:

python

```
#!/usr/bin/env python3
```

```
# Creating sets — removes duplicates automatically
```

```
failed_ips = {"192.168.1.10", "192.168.1.11", "192.168.1.10", "192.168.1.12"}  
print(failed_ips) # {'192.168.1.10', '192.168.1.11', '192.168.1.12'} — no duplicate  
  
# Adding and removing  
failed_ips.add("192.168.1.13")  
failed_ips.discard("192.168.1.10") # remove — no error if not present  
  
# Check membership — very fast  
if "192.168.1.11" in failed_ips:  
    print("This IP has failed logins")  
  
# Set operations — powerful for log analysis  
ssh_attempts = {"10.0.0.1", "10.0.0.2", "10.0.0.3", "10.0.0.4"}  
blocked_ips = {"10.0.0.3", "10.0.0.4", "10.0.0.5"}  
  
# IPs that attempted SSH AND are blocked  
both = ssh_attempts & blocked_ips  
print(f"Attempted and blocked: {both}")  
  
# IPs that attempted SSH but are NOT blocked  
not_blocked = ssh_attempts - blocked_ips  
print(f"Attempted but NOT blocked: {not_blocked}")  
  
# All unique IPs across both sets  
all_ips = ssh_attempts | blocked_ips  
print(f"All unique IPs: {all_ips}")
```

5. Choosing the Right Data Structure

Situation	Use
Ordered list of services to loop over	List
Server config with named fields	Dictionary
Fixed record that won't change	Tuple
Unique IPs from a log file	Set
Rows of tabular data	List of tuples
Nested config like server inventory	Dict of dicts

6. Real-World Example — Server Inventory Manager

python

```
#!/usr/bin/env python3

# =====

# Script Name: inventory_manager.py

# Description: Manage server inventory using

#     Python data structures

# Usage   : python3 inventory_manager.py

# =====

import subprocess

import sys

from datetime import datetime

# — Data Structure — Dict of Dicts ——————
INVENTORY = {

    "web01": {

        "ip"      : "192.168.1.10",

        "role"    : "web",

        "services": ["nginx", "php-fpm"],

        "port"    : 22,
    }
}
```

```
},
"web02": {
    "ip" : "192.168.1.11",
    "role" : "web",
    "services": ["nginx", "php-fpm"],
    "port" : 22,
},
"db01": {
    "ip" : "192.168.1.20",
    "role" : "database",
    "services": ["mysql", "rsyslog"],
    "port" : 22,
},
"cache01": {
    "ip" : "192.168.1.30",
    "role" : "cache",
    "services": ["redis", "rsyslog"],
    "port" : 22,
},
}
```

```
def ping_host(ip):
    """Return True if host responds to ping."""
    result = subprocess.run(
        ["ping", "-c", "1", "-W", "2", ip],
        capture_output=True
    )
    return result.returncode == 0
```

```
def get_servers_by_role(role):
    """Return list of server names matching a given role."""
    return [
        hostname
        for hostname, info in INVENTORY.items()
        if info["role"] == role
    ]

def get_all_unique_services():
    """Return sorted set of all unique services across all servers."""
    all_services = set()
    for info in INVENTORY.values():
        all_services.update(info["services"])
    return sorted(all_services)

def run_health_check():
    """Ping all servers and return results as a dict."""
    results = {}
    for hostname, info in INVENTORY.items():
        results[hostname] = {
            "ip" : info["ip"],
            "role" : info["role"],
            "reachable": ping_host(info["ip"])
        }
    return results

def print_inventory():
    """Print full inventory in a formatted table."""
    print(f"\n{'='*65}\n")
```

```

print(f" Server Inventory — {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}")
print(f"{'='*65}")

print(f" {'HOSTNAME':<12} {'IP':<16} {'ROLE':<12} {'SERVICES'}")
print(f" {'-'*60}")

for hostname, info in sorted(INVENTORY.items()):
    services_str = ", ".join(info["services"])

    print(f" {hostname:<12} {info['ip']:<16} {info['role']:<12} {services_str}")

print(f"{'='*65}")
print(f" Total servers: {len(INVENTORY)}")

# Group by role using a dict
roles = {}

for info in INVENTORY.values():
    role = info["role"]
    roles[role] = roles.get(role, 0) + 1

for role, count in sorted(roles.items()):
    print(f" {role.capitalize():<12}: {count} server(s)")

print(f"{'='*65}\n")

def print_health_check():
    """Run and print health check results."""
    print(f"\n{'='*50}")

    print(f" Health Check — {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}")
    print(f"{'='*50}")

    print(f" {'HOSTNAME':<12} {'IP':<16} {'STATUS'}")
    print(f" {'-'*45}")

```

```
results = run_health_check()

up_count = 0
down_count = 0

for hostname, data in sorted(results.items()):
    status = "UP" if data["reachable"] else "DOWN"

    if data["reachable"]:
        up_count += 1
    else:
        down_count += 1

    print(f" {hostname}<12} {data['ip']}<16} {status}")

print(f" {'-*45}'")
print(f" UP: {up_count} DOWN: {down_count}")
print(f"{'-*50}\n")

def main():
    print_inventory()

    print(" All unique services across inventory:")
    services = get_all_unique_services()
    for s in services:
        print(f" - {s}")

    print(f"\n Web servers: {get_servers_by_role('web')}")
    print(f" DB servers: {get_servers_by_role('database')}")

    print_health_check()
```

```
if __name__ == "__main__":
    main()
```

Key Takeaways from Module 4

- **Lists []** — ordered, changeable, for sequences you loop over
 - **Dictionaries {}** — key-value pairs, for named config and server data
 - **Tuples ()** — immutable records, for fixed data like server connections
 - **Sets {}** — unique values only, perfect for deduplicating log IPs
 - Use dict.get("key", default) for safe access without KeyError
 - Use "item" in list/dict/set to check existence — replaces grep in many cases
 - List comprehensions [x for x in list if condition] filter and transform in one line
 - Dict of dicts is the go-to pattern for server inventory and config management
-

Your Exercise — Module 4

Write a Python script called `log_analyzer.py` that:

1. Has a list of sample log lines hardcoded in the script (mix of INFO, WARNING, ERROR)
 2. Uses a dictionary to count how many of each log level appear
 3. Uses a set to collect all unique IP addresses mentioned in error lines
 4. Uses a list to store all ERROR lines for reporting
 5. Prints a formatted summary: total lines, count per level, unique IPs found
 6. Prints all ERROR lines in a separate section
-

Say "show solution" and we move to **Module 5 — Working with Files & the OS!** 

show solution and move to next module

10:50 PM

Solution — `log_analyzer.py`

python

```
#!/usr/bin/env python3

# =====

# Script Name: log_analyzer.py

# Description: Analyze log lines using Python

#     data structures

# Usage   : python3 log_analyzer.py

# =====

import re

from datetime import datetime

from collections import Counter


# — Sample Log Data ——————



LOG_LINES = [

    "2024-01-15 10:00:01 INFO Service nginx started successfully",

    "2024-01-15 10:00:05 INFO Service sshd started successfully",

    "2024-01-15 10:01:10 WARNING Disk usage at 78% on /var",

    "2024-01-15 10:02:15 ERROR Failed login attempt from 192.168.1.105",

    "2024-01-15 10:03:20 INFO Backup completed successfully",

    "2024-01-15 10:04:25 ERROR Connection refused from 10.0.0.44",

    "2024-01-15 10:05:30 WARNING Memory usage at 82%",

    "2024-01-15 10:06:35 ERROR Failed login attempt from 192.168.1.105",

    "2024-01-15 10:07:40 INFO User john logged in",

    "2024-01-15 10:08:45 ERROR Disk write failure on /dev/sdb from 10.0.0.55",

    "2024-01-15 10:09:50 WARNING CPU usage spike detected",

    "2024-01-15 10:10:55 ERROR Failed login attempt from 172.16.0.22",

    "2024-01-15 10:11:00 INFO Scheduled task completed",

    "2024-01-15 10:12:05 ERROR Service mysql crashed",

    "2024-01-15 10:13:10 INFO Log rotation completed",
```

]

— Functions ——————

```
def parse_log_lines(lines):
```

```
    """
```

Parse log lines into structured data.

Returns:

```
tuple: (level_counts dict, error_lines list, unique_ips set)
```

```
    """
```

```
level_counts = {} # dictionary — count per log level
```

```
error_lines = [] # list — all ERROR lines
```

```
unique_ips = set()# set — unique IPs from error lines
```

Regex pattern to extract IP addresses

```
ip_pattern = re.compile(r"\b(\d{1,3}(?:\.\d{1,3}){3})\b")
```

for line in lines:

```
    # Split line into parts — date, time, level, message
```

```
    parts = line.split()
```

```
    if len(parts) < 3:
```

```
        continue
```

```
    level = parts[2] # INFO, WARNING, ERROR
```

Count each log level using dictionary

```
    level_counts[level] = level_counts.get(level, 0) + 1
```

Collect ERROR lines and extract IPs

```

if level == "ERROR":
    error_lines.append(line)

    # Extract all IPs from this line using regex
    ips = ip_pattern.findall(line)
    unique_ips.update(ips) # add to set — duplicates ignored

return level_counts, error_lines, unique_ips

def print_summary(total, level_counts, unique_ips):
    """Print formatted analysis summary."""
    print(f"\n{'='*55}")
    print(f" Log Analysis Report — {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}")
    print(f"{'='*55}")
    print(f" Total lines analysed : {total}")
    print(f" {'-'*45}")

    # Print counts per level in a clean format
    for level in ["INFO", "WARNING", "ERROR"]:
        count = level_counts.get(level, 0)
        bar = "█" * count
        print(f" {level:<10}: {count:>3} {bar}")

    print(f" {'-'*45}")
    print(f" Unique IPs in errors : {len(unique_ips)}")
    for ip in sorted(unique_ips):
        print(f" → {ip}")
    print(f"{'='*55}")

```

```
def print_error_report(error_lines):
    """Print all ERROR lines in a separate section."""
    print(f"\n ⚠️ ERROR Lines ({len(error_lines)} total)")
    print("-"*50)
    for line in error_lines:
        # Highlight just the message part
        parts = line.split(None, 3)
        date = f"{parts[0]} {parts[1]}"
        message = parts[3] if len(parts) > 3 else ""
        print(f" [{date}] {message}")
    print(f"{'='*55}\n")
```

— Main ——————

```
def main():
    level_counts, error_lines, unique_ips = parse_log_lines(LOG_LINES)
    print_summary(len(LOG_LINES), level_counts, unique_ips)
    print_error_report(error_lines)
```

— Entry Point ——————

```
if __name__ == "__main__":
    main()
```

🔍 Explanation

Dictionary for counting log levels

python

```
level_counts[level] = level_counts.get(level, 0) + 1
```

dict.get(key, 0) returns the current count or 0 if the key doesn't exist yet, then adds 1. This is the cleanest Python pattern for counting occurrences — replaces a manual if key in dict check.

Set for unique IPs

```
python
```

```
unique_ips.update(ips)
```

`set.update()` adds multiple items at once and automatically ignores duplicates. So if 192.168.1.105 appears in 3 error lines, it only appears once in the set — no extra logic needed.

Regex to extract IPs

```
python
```

```
ip_pattern = re.compile(r"\b(\d{1,3}(?:\.\d{1,3}){3})\b")
```

```
ips = ip_pattern.findall(line)
```

`re.compile()` builds a reusable pattern. `findall()` returns all matches in a line as a list. This replaces grep -oE '[0-9]+\.[0-9]+\.[0-9]+\.[0-9]+' from Bash but gives us a Python list directly.

Unpacking log line parts

```
python
```

```
parts = line.split(None, 3)
```

```
---
```

``split(None, 3)`` splits on any whitespace but stops after 3 splits — so the message part stays intact even if it contains spaces. The `3` is the max splits, giving us `[date, time, level, full_message]`.

```
---
```

```
## 📈 Expected Output
```

```
---
```

```
=====
```

```
Log Analysis Report — 2024-01-15 10:35:22
```

```
=====
```

```
Total lines analysed : 15
```

```
-----
```

```
INFO    : 7 [REDACTED]
```

WARNING : 3 [REDACTED]

ERROR : 5 [REDACTED]

Unique IPs in errors : 4

→ 10.0.0.44

→ 10.0.0.55

→ 172.16.0.22

→ 192.168.1.105

⚠ ERROR Lines (5 total)

[2024-01-15 10:02:15] Failed login attempt from 192.168.1.105

[2024-01-15 10:04:25] Connection refused from 10.0.0.44

[2024-01-15 10:06:35] Failed login attempt from 192.168.1.105

[2024-01-15 10:08:45] Disk write failure on /dev/sdb from 10.0.0.55

[2024-01-15 10:10:55] Failed login attempt from 172.16.0.22

Module 5 — Working with Files & the OS

This is where Python truly replaces Bash for sysadmin work. Python gives you clean, cross-platform, error-safe ways to work with files, directories, and the operating system.

1. Reading Files

python

```
#!/usr/bin/env python3
```

```
# — Method 1: Read entire file at once —————
```

```
with open("/etc/hostname", "r") as f:
```

```
content = f.read()
print(content.strip())

# — Method 2: Read line by line ————
# Equivalent of: while IFS= read -r line; do ... done < file
with open("/etc/passwd", "r") as f:
    for line in f:
        line = line.strip()
        if line.startswith("root"):
            print(f"Found root entry: {line}")

# — Method 3: Read all lines into a list ————
with open("/etc/hosts", "r") as f:
    lines = f.readlines() # list of lines with \n included

# Strip newlines from all lines at once
lines = [line.strip() for line in lines]
print(f"Hosts file has {len(lines)} lines")

# — Method 4: Read and parse a config file ————
config = {}
with open("/etc/os-release", "r") as f:
    for line in f:
        line = line.strip()
        if "=" in line and not line.startswith("#"):
            key, value = line.split("=", 1)
            config[key] = value.strip("''")

print(f"OS Name: {config.get('PRETTY_NAME', 'Unknown')}")
```

2. Writing Files

```
python
```

```
#!/usr/bin/env python3  
from datetime import datetime
```

```
# — Write a new file (overwrites if exists) —
```

```
with open("/tmp/report.txt", "w") as f:  
    f.write("Server Health Report\n")  
    f.write(f"Generated: {datetime.now()}\n")  
    f.write("=" * 40 + "\n")
```

```
# — Append to existing file ——————
```

```
# Equivalent of >> in Bash  
with open("/tmp/report.txt", "a") as f:  
    f.write("New line appended\n")
```

```
# — Write multiple lines at once ——————
```

```
lines = [  
    "nginx : RUNNING\n",  
    "sshd : RUNNING\n",  
    "mysql : STOPPED\n",  
]  
with open("/tmp/services.txt", "w") as lines_file:  
    lines_file.writelines(lines)
```

```
# — Write formatted report safely ——————
```

```
def write_report(filepath, data):  
    """Write a report file with error handling."""
```

```

try:
    with open(filepath, "w") as f:
        f.write(f"Report Generated: {datetime.now()}\n")
        f.write("=" * 40 + "\n")
        for key, value in data.items():
            f.write(f"{key:<20}: {value}\n")
        print(f"Report written to {filepath}")

except PermissionError:
    print(f"Error: No permission to write to {filepath}")

except Exception as e:
    print(f"Error writing report: {e}")

write_report("/tmp/audit.txt", {"Hostname": "web01", "Status": "OK"})

```

3. The os Module — Navigating the Filesystem

```

python
#!/usr/bin/env python3

import os

# — Path operations ——————
print(os.getcwd())          # current directory — like pwd
os.chdir("/tmp")           # change directory — like cd
print(os.path.exists("/etc/passwd"))  # True/False — like [ -e ]
print(os.path.isfile("/etc/passwd"))   # True/False — like [ -f ]
print(os.path.isdir("/var/log"))      # True/False — like [ -d ]
print(os.path.getsize("/etc/passwd")) # file size in bytes
print(os.path.basename("/var/log/sys.log")) # sys.log — like basename
print(os.path.dirname("/var/log/sys.log")) # /var/log — like dirname
print(os.path.join("/var", "log", "sys.log")) # /var/log/sys.log — safe path join

```

```
# — Directory operations ——————  
os.makedirs("/tmp/myapp/logs", exist_ok=True) # mkdir -p equivalent  
os.rename("/tmp/old.txt", "/tmp/new.txt") # mv equivalent  
os.remove("/tmp/file.txt") # rm equivalent  
os.rmdir("/tmp/empty_dir") # rmdir equivalent
```

```
# — List directory contents ——————  
# Like ls /var/log/*.log  
log_files = os.listdir("/var/log")  
print(log_files)
```

```
# — Walk entire directory tree ——————  
# Like find /var/log -type f  
for dirname, dirnames, filenames in os.walk("/var/log"):  
    for filename in filenames:  
        full_path = os.path.join(dirname, filename)  
        size = os.path.getsize(full_path)  
        print(f"{full_path} — {size} bytes")
```

```
# — Environment variables ——————  
home = os.environ.get("HOME", "/root") # like $HOME in Bash  
path = os.environ.get("PATH", "")  
user = os.environ.get("USER", "unknown")  
print(f"Home: {home} | User: {user}")
```

4. The `pathlib` Module — Modern Path Handling

```
python  
#!/usr/bin/env python3
```

```
from pathlib import Path

# pathlib is the modern way to work with paths
log_dir = Path("/var/log")

# Check existence
print(log_dir.exists())      # True/False
print(log_dir.is_dir())      # True/False

# Build paths safely — no string concatenation needed
log_file = log_dir / "syslog"      # /var/log/syslog
backup  = log_dir / "archive" / "old" # /var/log/archive.old

# Create directories
backup.mkdir(parents=True, exist_ok=True)

# Find files — like find command
log_files = list(log_dir.glob("*.log")) # all .log files
all_files = list(log_dir.rglob("*.gz")) # recursive — all .gz files

# File info
for f in log_files[:5]:
    print(f"Name: {f.name} | Size: {f.stat().st_size} bytes")

# Read and write
config = Path("/etc/hostname")
hostname = config.read_text().strip()
print(f"Hostname: {hostname}")
```

```
Path("/tmp/test.txt").write_text("Hello from pathlib!\n")
```

5. The shutil Module — File Operations

```
python
```

```
#!/usr/bin/env python3
```

```
import shutil
```

```
from pathlib import Path
```

```
# Copy a file — like cp
```

```
shutil.copy("/etc/hosts", "/tmp/hosts.bak")
```

```
# Copy with metadata — like cp -p
```

```
shutil.copy2("/etc/hosts", "/tmp/hosts.bak")
```

```
# Copy entire directory — like cp -r
```

```
shutil.copytree("/etc/nginx", "/tmp/nginx_backup")
```

```
# Move a file or directory — like mv
```

```
shutil.move("/tmp/old_name.txt", "/tmp/new_name.txt")
```

```
# Delete entire directory tree — like rm -rf
```

```
shutil.rmtree("/tmp/old_backup", ignore_errors=True)
```

```
# Check available disk space
```

```
total, used, free = shutil.disk_usage("/")
```

```
print(f"Total : {total // (2**30)} GB")
```

```
print(f"Used : {used // (2**30)} GB")
```

```
print(f"Free : {free // (2**30)} GB")
```

```
print(f"Usage : {used / total * 100:.1f}%")
```

6. The subprocess Module — Running System Commands

```
python  
#!/usr/bin/env python3  
import subprocess  
  
# — Run a command and get output _____  
# Replaces: OUTPUT=$(command) in Bash  
result = subprocess.run(  
    ["df", "-h", "/"],  
    capture_output=True,  
    text=True  
)  
print(result.stdout)  
print(result.returncode) # 0 = success  
  
# — Check if command succeeded _____  
result = subprocess.run(["systemctl", "is-active", "nginx"],  
    capture_output=True, text=True)  
if result.returncode == 0:  
    print("nginx is running")  
else:  
    print("nginx is NOT running")  
  
# — Run shell pipeline _____  
# Replaces: ps aux | grep nginx | wc -l  
result = subprocess.run(  
    "ps aux | grep nginx | wc -l",  
    shell=True,
```

```

capture_output=True,
text=True
)

print(f"nginx processes: {result.stdout.strip()}")

# — Run and raise exception on failure ————
try:
    subprocess.run(["useradd", "testuser"], check=True,
                  capture_output=True, text=True)
    print("User created")
except subprocess.CalledProcessError as e:
    print(f"Command failed: {e.stderr}")

# — Run with timeout ————
try:
    result = subprocess.run(
        ["ping", "-c", "5", "8.8.8.8"],
        capture_output=True,
        text=True,
        timeout=10 # fail if takes more than 10 seconds
    )
except subprocess.TimeoutExpired:
    print("Command timed out!")

```

7. Real-World Example — File & OS Automation Script

```

python
#!/usr/bin/env python3
# =====
# Script Name: file_manager.py

```

```
# Description: Automated file management —  
#           backup, cleanup, disk reporting  
# Usage    : sudo python3 file_manager.py  
# =====
```

```
import os  
import sys  
import shutil  
import logging  
import subprocess  
from pathlib import Path  
from datetime import datetime
```

```
# — Setup ——————  
logging.basicConfig(  
    level=logging.INFO,  
    format="[%(asctime)s] [%(levelname)s] %(message)s",  
    datefmt="%Y-%m-%d %H:%M:%S",  
    handlers=[logging.StreamHandler(sys.stdout)]  
)  
log = logging.getLogger()
```

```
BACKUP_DIR  = Path("/backup")  
LOG_DIR     = Path("/var/log")  
DISK_THRESHOLD = 85  
MAX_LOG_SIZE = 100 * 1024 * 1024 # 100MB in bytes
```

```
# — Functions ——————  
def check_disk_space(path="/"):
```

```
"""Return disk usage percentage for a path."""
total, used, free = shutil.disk_usage(path)

return round(used / total * 100, 1)

def get_large_files(directory, max_size_bytes):
    """Return list of files larger than max_size_bytes."""
    large_files = []
    for filepath in Path(directory).rglob("*"):
        if filepath.is_file():
            try:
                size = filepath.stat().st_size
                if size > max_size_bytes:
                    large_files.append((filepath, size))
            except PermissionError:
                continue
    return sorted(large_files, key=lambda x: x[1], reverse=True)

def backup_file(source, dest_dir):
    """Backup a file with timestamp in filename."""
    source = Path(source)
    dest_dir = Path(dest_dir)
    dest_dir.mkdir(parents=True, exist_ok=True)

    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    dest_file = dest_dir / f"{source.stem}_{timestamp}{source.suffix}"

    shutil.copy2(source, dest_file)
    log.info(f"Backed up: {source} → {dest_file}")
    return dest_file
```

```
def cleanup_old_files(directory, days=30):
    """Remove files older than N days."""
    import time
    cutoff = time.time() - (days * 86400)
    removed = 0

    for filepath in Path(directory).rglob("*.gz"):
        if filepath.stat().st_mtime < cutoff:
            filepath.unlink()
            log.info(f"Removed old file: {filepath}")
            removed += 1

    return removed

def generate_disk_report():
    """Generate disk usage report using shutil and subprocess."""
    report = {}

    # Check key mount points
    for mount in ["/", "/home", "/var", "/tmp"]:
        if os.path.exists(mount):
            total, used, free = shutil.disk_usage(mount)
            report[mount] = {
                "total_gb": round(total / 2**30, 1),
                "used_gb": round(used / 2**30, 1),
                "free_gb": round(free / 2**30, 1),
                "percent": round(used / total * 100, 1)
            }
    
```

```

return report

def print_disk_report(report):
    """Print formatted disk report."""
    print(f"\n{'='*65}")
    print(f" Disk Usage Report — {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}")
    print(f"{'='*65}")
    print(f" {'MOUNT':<10} {'TOTAL':>8} {'USED':>8} {'FREE':>8} {'USE%':>6}")
    print(f" {'-'*55}")

for mount, data in report.items():
    flag = "⚠️" if data["percent"] >= DISK_THRESHOLD else ""
    print(f" {mount:<10} "
          f"{str(data['total_gb'])+'G':>8} "
          f"{str(data['used_gb'])+'G':>8} "
          f"{str(data['free_gb'])+'G':>8} "
          f"{str(data['percent'])+'%':>6}{flag}")

    print(f"{'='*65}\n")

def main():
    log.info("===== File Manager Started =====")
    # Disk report
    report = generate_disk_report()
    print_disk_report(report)

    # Find large log files

```

```

log.info(f"Scanning {LOG_DIR} for large files...")
large_files = get_large_files(LOG_DIR, MAX_LOG_SIZE)

if large_files:
    log.warning(f"Found {len(large_files)} large log file(s):")
    for filepath, size in large_files:
        size_mb = round(size / 1024 / 1024, 1)
        log.warning(f" {filepath} — {size_mb}MB")
else:
    log.info("No oversized log files found")

# Cleanup old backups
if BACKUP_DIR.exists():
    removed = cleanup_old_files(BACKUP_DIR, days=30)
    log.info(f"Cleanup complete — removed {removed} old backup(s)")

log.info("===== File Manager Completed =====")

if __name__ == "__main__":
    main()

```

Key Takeaways from Module 5

- Always use `with open()` for file operations — it closes files safely automatically
- Use "r" to read, "w" to write (overwrite), "a" to append
- `os.path` gives you Bash-equivalent file checks — `exists()`, `isfile()`, `isdir()`
- `pathlib.Path` is the modern way — use `/` operator to join paths safely
- `shutil` handles copy, move, delete, and disk usage — replaces `cp`, `mv`, `rm -rf`
- `subprocess.run()` with `check=True` raises an exception on failure — like `set -e`
- Use `capture_output=True`, `text=True` to get command output as a string

- Never use shell=True unless you absolutely need pipeline syntax — it is a security risk
-

Your Exercise — Module 5

Write a Python script called config_backup.py that:

1. Accepts a config file path as a command line argument
 2. Checks the file exists — if not, print error and exit
 3. Reads the file and counts how many lines are comments #, blank, and active config lines
 4. Creates a timestamped backup copy in /tmp/config_backups/
 5. Writes a small report file alongside the backup showing the line count breakdown
 6. Prints disk usage of /tmp before and after the backup
-

Say "show solution" and we move straight to **Module 6 — Error Handling!** 

show solution and move to next module

11:30 PM

Solution — config_backup.py

python

```
#!/usr/bin/env python3
# =====
# Script Name: config_backup.py
# Description: Backup a config file with
#               analysis and disk reporting
# Usage   : python3 config_backup.py <config_file>
# =====
```

```
import os
import sys
import shutil
import logging
from pathlib import Path
```

```
from datetime import datetime

# — Setup ——————
logging.basicConfig(
    level=logging.INFO,
    format="[%(asctime)s] [%(levelname)s] %(message)s",
    datefmt="%Y-%m-%d %H:%M:%S",
    handlers=[logging.StreamHandler(sys.stdout)]
)
log = logging.getLogger()
```

```
BACKUP_DIR = Path("/tmp/config_backups")
```

```
# — Functions ——————
```

```
def validate_args():
    """Validate and return config file path from arguments."""
    if len(sys.argv) < 2:
        log.error(f"Usage: {sys.argv[0]} <config_file>")
        sys.exit(1)
```

```
config_path = Path(sys.argv[1])
```

```
if not config_path.exists():
    log.error(f"File not found: {config_path}")
    sys.exit(1)
```

```
if not config_path.is_file():
    log.error(f"Not a file: {config_path}")
    sys.exit(1)
```

```

return config_path

def get_disk_usage_percent(path="/tmp"):
    """Return disk usage percentage for a given path."""
    total, used, free = shutil.disk_usage(path)
    return round(used / total * 100, 1)

def analyze_config(filepath):
    """
    Read config file and categorise each line.

    Returns:
        dict with counts of comment, blank, and active lines
        and the full list of lines
    """
    counts = {"comment": 0, "blank": 0, "active": 0, "total": 0}
    lines = []

    with open(filepath, "r") as f:
        for line in f:
            raw = line.rstrip("\n")
            lines.append(raw)
            stripped = raw.strip()
            counts["total"] += 1

            if not stripped:
                counts["blank"] += 1
            elif stripped.startswith("#"):

```

```
        counts["comment"] += 1

    else:

        counts["active"] += 1


    return counts, lines


def create_backup(source_path):
    """
    Create a timestamped backup of the config file.

    Returns:
        Path to the backup file
    """

    BACKUP_DIR.mkdir(parents=True, exist_ok=True)

    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    backup_name = f"{source_path.stem}_{timestamp}{source_path.suffix}"
    backup_path = BACKUP_DIR / backup_name

    shutil.copy2(source_path, backup_path)
    log.info(f"Backup created: {backup_path}")

    return backup_path


def write_report(backup_path, source_path, counts):
    """
    Write a report file alongside the backup.

    report_path = backup_path.with_suffix(".report.txt")

    with open(report_path, "w") as f:
        f.write("=" * 45 + "\n")

```

```

f.write("Config File Backup Report\n")
f.write("=" * 45 + "\n")

f.write(f"{'Source file':<20}: {source_path}\n")
f.write(f"{'Backup file':<20}: {backup_path}\n")
f.write(f"{'Generated':<20}: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}\n")
f.write(f"{'File size':<20}: {source_path.stat().st_size} bytes\n")

f.write("-" * 45 + "\n")

f.write("Line Analysis:\n")
f.write(f" {'Total lines':<18}: {counts['total']}\n")
f.write(f" {'Active config':<18}: {counts['active']}\n")
f.write(f" {'Comments (#)':<18}: {counts['comment']}\n")
f.write(f" {'Blank lines':<18}: {counts['blank']}\n")

f.write("=" * 45 + "\n")

log.info(f"Report written : {report_path}")

return report_path

def print_analysis(source_path, counts):
    """Print formatted analysis to console."""
    print("\n'*50")
    print(" Config File Analysis")
    print('*50')
    print(f" {'File':<18}: {source_path}")
    print(f" {'Size':<18}: {source_path.stat().st_size} bytes")
    print('*42')
    print(f" {'Total lines':<18}: {counts['total']}")
    print(f" {'Active config':<18}: {counts['active']}")
    print(f" {'Comments (#)':<18}: {counts['comment']}")
    print(f" {'Blank lines':<18}: {counts['blank']}")

```

```
print(f'{"*50}\n')

# — Main ——————
def main():

    config_path = validate_args()

    # Disk usage BEFORE
    disk_before = get_disk_usage_percent("/tmp")
    log.info(f'Disk usage before backup: {disk_before}%')

    # Analyse the config file
    log.info(f'Analysing: {config_path}')
    counts, lines = analyze_config(config_path)
    print_analysis(config_path, counts)

    # Create backup
    backup_path = create_backup(config_path)

    # Write report
    write_report(backup_path, config_path, counts)

    # Disk usage AFTER
    disk_after = get_disk_usage_percent("/tmp")
    log.info(f'Disk usage after backup: {disk_after}%')
    log.info(f'Disk change: {round(disk_after - disk_before, 2)}%')
    log.info("===== Config Backup Complete =====")

# — Entry Point ——————
if __name__ == "__main__":
```

```
main()
```

Explanation

Path.with_suffix() for clean filename manipulation

python

```
report_path = backup_path.with_suffix(".report.txt")
```

pathlib lets you swap file extensions cleanly.

If `backup_path` is `/tmp/config_backups/nginx_20240115.conf`, this produces `/tmp/config_backups/nginx_20240115.report.txt` — no string slicing or sed needed.

Categorising lines with strip() and startswith()

python

```
if not stripped:
```

```
    counts["blank"] += 1
```

```
elif stripped.startswith("#"):
```

```
    counts["comment"] += 1
```

```
else:
```

```
    counts["active"] += 1
```

`strip()` removes all leading and trailing whitespace. An empty string after `strip` means a blank line. `startswith("#")` replaces grep `^#`. This is clean, readable, and requires zero external commands.

shutil.copy2() vs shutil.copy()

python

```
shutil.copy2(source_path, backup_path)
```

```
...
```

`copy2` preserves the original file's metadata including timestamps — equivalent of `cp -p` in Bash. Always use `copy2` for backups so you keep the original modification time.

```
---
```

📄 Expected Output

[2024-01-15 10:30:01] [INFO] Disk usage before backup: 12.4%

[2024-01-15 10:30:01] [INFO] Analysing: /etc/ssh/sshd_config

=====

Config File Analysis

=====

File : /etc/ssh/sshd_config

Size : 3453 bytes

Total lines : 145

Active config : 12

Comments (#) : 89

Blank lines : 44

=====

[2024-01-15 10:30:01] [INFO] Backup created:

/tmp/config_backups/sshd_config_20240115_103001.conf

[2024-01-15 10:30:01] [INFO] Report written:

/tmp/config_backups/sshd_config_20240115_103001.report.txt

[2024-01-15 10:30:01] [INFO] Disk usage after backup: 12.4%

[2024-01-15 10:30:01] [INFO] Disk change: 0.0%

[2024-01-15 10:30:01] [INFO] ===== Config Backup Complete =====

Module 6 — Error Handling

In Bash you used set -euo pipefail and trap. Python has a far more powerful and expressive system called **exceptions**. This is what makes Python scripts truly production-safe.

1. The Problem — Why Error Handling Matters

```
python
#!/usr/bin/env python3

# Without error handling — script crashes with ugly traceback
with open("/etc/shadow", "r") as f:  # PermissionError if not root
    content = f.read()

# Output without error handling:
# Traceback (most recent call last):
#   File "script.py", line 4, in <module>
#     with open("/etc/shadow", "r") as f:
#       ^
#   PermissionError: [Errno 13] Permission denied: '/etc/shadow'
```

2. try / except — The Foundation

```
python
#!/usr/bin/env python3

# Basic structure — equivalent of checking $? after every command in Bash
try:
    # Code that might fail
    with open("/etc/shadow", "r") as f:
        content = f.read()
    print("File read successfully")

except PermissionError:
    print("Error: Permission denied — run as root")

except FileNotFoundError:
    print("Error: File not found")
```

```
except Exception as e:  
    print(f"Unexpected error: {e}")
```

3. Common Exception Types for Sysadmins

python

```
#!/usr/bin/env python3
```

```
import subprocess
```

```
# — File & OS exceptions ——————
```

```
try:  
    with open("/nonexistent/file.txt") as f:  
        f.read()  
except FileNotFoundError as e:  
    print(f"File not found: {e}")
```

try:

```
    with open("/etc/shadow") as f:  
        f.read()  
except PermissionError:  
    print("Permission denied — need root")
```

try:

```
    os.remove("/tmp/doesnt_exist.txt")  
except FileNotFoundError:  
    print("Nothing to delete — file already gone")
```

```
# — Value & Type exceptions ——————
```

try:

```

threshold = int("abc")      # can't convert string to int

except ValueError as e:
    print(f"Invalid value: {e}")

try:
    result = 100 / 0        # division by zero

except ZeroDivisionError:
    print("Cannot divide by zero")

# — Network & subprocess exceptions ——————
try:
    result = subprocess.run(
        ["systemctl", "restart", "nginx"],
        check=True,
        capture_output=True,
        text=True
    )

except subprocess.CalledProcessError as e:
    print(f"Command failed with exit code {e.returncode}")
    print(f"Error output: {e.stderr}")

except FileNotFoundError:
    print("systemctl not found — not a systemd system?")

```

4. try / except / else / finally

python

```
#!/usr/bin/env python3
```

```
# Full structure — more control than Bash trap
```

```

try:
    # Attempt the risky operation
    with open("/var/log/syslog", "r") as f:
        lines = f.readlines()

except FileNotFoundError:
    print("Log file not found")
    lines = []

except PermissionError:
    print("Permission denied reading log")
    lines = []

else:
    # Runs ONLY if try succeeded — no exception occurred
    print(f"Successfully read {len(lines)} lines")

finally:
    # ALWAYS runs — like trap EXIT in Bash
    print("Log reading attempt complete")

```

Block	When it runs
try	Always attempted
except	Only if an exception occurred
else	Only if NO exception occurred
finally	Always — with or without exception

5. Raising Your Own Exceptions

python

```
#!/usr/bin/env python3
```

```

def validate_threshold(value):
    """Validate a threshold value is between 0 and 100."""
    if not isinstance(value, (int, float)):
        raise TypeError(f"Threshold must be a number, got {type(value)}")
    if value < 0 or value > 100:
        raise ValueError(f"Threshold must be between 0-100, got {value}")
    return value

def validate_username(username):
    """Validate a username string."""
    if not username:
        raise ValueError("Username cannot be empty")
    if not username.isalnum() and not all(c in "-_" for c in username if not c.isalnum()):
        raise ValueError(f"Invalid username: {username}")
    return username

# Use them
try:
    threshold = validate_threshold(110)
except ValueError as e:
    print(f"Validation error: {e}")

try:
    username = validate_username("")
except ValueError as e:
    print(f"Validation error: {e}")

```

6. Custom Exception Classes

```
python

#!/usr/bin/env python3


# Define custom exceptions for your application

class SysAdminError(Exception):

    """Base exception for sysadmin scripts."""

    pass


class ServiceError(SysAdminError):

    """Raised when a service operation fails."""

    pass


class DiskSpaceError(SysAdminError):

    """Raised when disk space is critically low."""

    pass


class ConfigError(SysAdminError):

    """Raised when a config file is invalid."""

    pass


# Use them

def check_disk_space(path, minimum_gb=1):

    import shutil

    total, used, free = shutil.disk_usage(path)

    free_gb = free / 2**30

    if free_gb < minimum_gb:

        raise DiskSpaceError(

            f"Only {free_gb:.1f}GB free on {path} — "

            f"minimum required: {minimum_gb}GB")
```

```

    )
return free_gb

def restart_service(service):
    import subprocess
    result = subprocess.run(
        ["systemctl", "restart", service],
        capture_output=True, text=True
    )
    if result.returncode != 0:
        raise ServiceError(
            f"Failed to restart {service}: {result.stderr.strip()}"
        )

# Catch specific custom exceptions
try:
    check_disk_space("/", minimum_gb=5)
    restart_service("nginx")
except DiskSpaceError as e:
    print(f"DISK ERROR: {e}")
except ServiceError as e:
    print(f"SERVICE ERROR: {e}")
except SysAdminError as e:
    print(f"SYSADMIN ERROR: {e}")

```

7. Context Managers — The Python trap

```

python
#!/usr/bin/env python3
import tempfile

```

```

import os

from contextlib import contextmanager


# — Basic context manager — auto cleanup ————
@contextmanager
def temp_file(suffix=".tmp"):

    """
    Create a temp file and guarantee cleanup on exit.

    Equivalent of TMPFILE=$(mktemp) + trap cleanup EXIT in Bash.
    """

    tmpfile = tempfile.mktemp(suffix=suffix)

    try:
        print(f"Created temp file: {tmpfile}")
        yield tmpfile      # script runs here
    finally:
        if os.path.exists(tmpfile):
            os.remove(tmpfile)
        print(f"Cleared up: {tmpfile}")

    # Use it — file is ALWAYS cleaned up even if exception occurs
    with temp_file(".sh") as tmpf:
        with open(tmpf, "w") as f:
            f.write("#!/bin/bash\necho hello\n")
        print(f"Working with: {tmpf}")
        # tmpf is automatically deleted when this block exits

```

8. The logging Module with Error Levels

python

```
#!/usr/bin/env python3
```

```
import logging
import sys

def setup_logger(logfile="/var/log/myscript.log", level=logging.INFO):
    """Production-ready logger with file and console output."""
    logger = logging.getLogger("sysadmin_script")
    logger.setLevel(level)

    formatter = logging.Formatter(
        "[%(asctime)s] [%(levelname)s] %(message)s",
        datefmt="%Y-%m-%d %H:%M:%S"
    )

    # Console — INFO and above
    console = logging.StreamHandler(sys.stdout)
    console.setLevel(logging.INFO)
    console.setFormatter(formatter)

    # File — DEBUG and above (captures everything)
    try:
        file_handler = logging.FileHandler(logfile)
        file_handler.setLevel(logging.DEBUG)
        file_handler.setFormatter(formatter)
        logger.addHandler(file_handler)
    except PermissionError:
        print(f"Warning: Cannot write to {logfile} — logging to console only")

    logger.addHandler(console)
    return logger
```

```
log = setup_logger()

# Usage
log.debug("Variable value: x=42")      # only goes to file
log.info("Script started")
log.warning("Disk usage above 80%")
log.error("Service failed to start")
log.critical("System out of disk space")

# Log exceptions with full traceback
try:
    result = 1 / 0
except ZeroDivisionError:
    log.exception("Unexpected math error") # logs full traceback automatically
```

9. Production-Grade Script with Full Error Handling

```
python
#!/usr/bin/env python3
# =====
# Script Name: safe_backup.py
# Description: Production backup script with
#     full Python error handling
# Usage   : sudo python3 safe_backup.py <source_dir>
# =====

import os
import sys
import shutil
```

```
import logging
import tempfile
from pathlib import Path
from datetime import datetime
from contextlib import contextmanager

# — Custom Exceptions ——————
class BackupError(Exception):
    pass

class DiskSpaceError(BackupError):
    pass

class SourceError(BackupError):
    pass

# — Setup ——————
logging.basicConfig(
    level=logging.INFO,
    format="[%(asctime)s] [%(levelname)s] %(message)s",
    datefmt="%Y-%m-%d %H:%M:%S",
    handlers=[
        logging.StreamHandler(sys.stdout),
        logging.FileHandler("/var/log/backup.log")
    ]
)
log = logging.getLogger()

BACKUP_DIR = Path("/backup")
```

```
RETENTION_DAYS = 7
```

```
# — Context Manager ——————
```

```
@contextmanager
```

```
def temp_backup_file(suffix=".tar.gz"):
```

```
    """Temp file that is always cleaned up — like trap cleanup EXIT."""
```

```
    tmpfile = Path(tempfile.mktemp(suffix=suffix))
```

```
    try:
```

```
        yield tmpfile
```

```
    finally:
```

```
        if tmpfile.exists():
```

```
            tmpfile.unlink()
```

```
            log.debug(f"Cleaned up temp file: {tmpfile}")
```

```
# — Functions ——————
```

```
def check_root():
```

```
    if os.geteuid() != 0:
```

```
        raise PermissionError("Must run as root. Use sudo.")
```



```
def validate_source(source_str):
```

```
    source = Path(source_str)
```

```
    if not source.exists():
```

```
        raise SourceError(f"Source not found: {source}")
```

```
    if not source.is_dir():
```

```
        raise SourceError(f"Source is not a directory: {source}")
```

```
    return source
```

```
def check_disk_space(source, dest):
```

```
    source_size = sum(
```

```

f.stat().st_size

for f in source.glob("*")
    if f.is_file()
)
    _, free = shutil.disk_usage(dest)

if source_size >= free:
    raise DiskSpaceError(
        f"Not enough space. Need: {source_size // 2**20}MB "
        f"Available: {free // 2**20}MB"
    )

def create_backup(source, dest_dir):
    dest_dir.mkdir(parents=True, exist_ok=True)
    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    final_dest = dest_dir / f"{source.name}_{timestamp}.tar.gz"

    with temp_backup_file() as tmp:
        log.info(f"Compressing {source} ...")
        shutil.make_archive(
            str(tmp.with_suffix("").with_suffix("")),
            "gztar",
            source.parent,
            source.name
        )
        tmp_gz = tmp.with_suffix("").with_suffix(") \
            .parent / (tmp.stem.rstrip(".") + ".tar.gz")
        shutil.move(str(tmp_gz), str(final_dest))

    log.info(f"Backup saved : {final_dest}")

```

```

log.info(f"Backup size : {final_dest.stat().st_size // 2**20}MB")

return final_dest


def apply_retention(dest_dir):
    import time

    cutoff = time.time() - (RETENTION_DAYS * 86400)

    removed = 0

    for f in dest_dir.glob("*.tar.gz"):
        if f.stat().st_mtime < cutoff:
            f.unlink()
            log.info(f"Removed old backup: {f.name}")
            removed += 1

    log.info(f"Retention policy applied — {removed} old backup(s) removed")


# — Main ——————
def main():

    try:
        check_root()

        if len(sys.argv) < 2:
            raise ValueError(f"Usage: {sys.argv[0]} <source_directory>")

        source = validate_source(sys.argv[1])
        dest = BACKUP_DIR / source.name

        log.info("===== Backup Started =====")
        check_disk_space(source, BACKUP_DIR if BACKUP_DIR.exists() else Path("/"))

        backup_file = create_backup(source, dest)
        apply_retention(dest)
    
```

```
log.info("===== Backup Completed Successfully =====")

except PermissionError as e:
    log.error(f"Permission error: {e}")
    sys.exit(1)

except (SourceError, DiskSpaceError) as e:
    log.error(f"Backup error: {e}")
    sys.exit(1)

except ValueError as e:
    log.error(str(e))
    sys.exit(1)

except Exception as e:
    log.exception(f"Unexpected error: {e}")
    sys.exit(1)

# — Entry Point ——————
if __name__ == "__main__":
    main()
```

Key Takeaways from Module 6

- Use try/except around any code that can fail — file ops, subprocess calls, conversions
- Catch specific exceptions first (FileNotFoundException) before generic (Exception)
- Use finally for cleanup — it always runs just like trap EXIT in Bash
- Use else in try blocks — it only runs when no exception occurred
- Raise exceptions with raise ValueError("message") to signal errors clearly
- Build custom exception classes for your application — makes error handling readable

- Use `@contextmanager` to build safe resource managers — the Python trap pattern
 - Use `log.exception()` inside `except` blocks — it captures the full traceback automatically
 - Never use bare `except:` without a type — it catches everything including keyboard interrupts
-

Your Exercise — Module 6

Write a Python script called `safe_user_create.py` that:

1. Defines custom exceptions: `UserExistsError`, `InvalidUsernameError`, `PermissionError`
 2. Has a `validate_username()` function that raises `InvalidUsernameError` for bad input
 3. Has a `create_user()` function wrapped in full `try/except/else/finally`
 4. Logs all operations with the logging module
 5. Uses a context manager to write a temp audit log that is copied to `/var/log/` on success or deleted on failure
 6. The `main()` catches all custom exceptions separately and exits with code 1 on any failure
-

Say "show solution" and we move to **Module 7 — Sysadmin Libraries!** 

show solution and move to next module

11:36 PM

Solution — `safe_user_create.py`

python

```
#!/usr/bin/env python3
# =====
# Script Name: safe_user_create.py
# Description: Safely create a system user
#           with full error handling
# Usage   : sudo python3 safe_user_create.py <username>
# =====
```

`import os`

`import sys`

```
import subprocess
import logging
import tempfile
import shutil
from pathlib import Path
from datetime import datetime
from contextlib import contextmanager

# — Custom Exceptions ——————
class SysAdminError(Exception):
    """Base exception for all sysadmin errors."""
    pass

class UserExistsError(SysAdminError):
    """Raised when user already exists."""
    pass

class InvalidUsernameError(SysAdminError):
    """Raised when username fails validation."""
    pass

class UserCreationError(SysAdminError):
    """Raised when useradd command fails."""
    pass

# — Logger Setup ——————
logging.basicConfig(
    level=logging.INFO,
    format='[%(asctime)s] [%(levelname)s] %(message)s',
```

```
datefmt="%Y-%m-%d %H:%M:%S",
handlers=[logging.StreamHandler(sys.stdout)]
)

log = logging.getLogger()
```

```
AUDIT_LOG_DIR = Path("/var/log")
```

```
# — Context Manager -----
```

```
@contextmanager
```

```
def audit_log_session(username):
```

```
"""
```

```
    Write audit events to a temp file.
```

```
    On success: copy to /var/log/user_create_audit.log
```

```
    On failure: delete temp file cleanly.
```

```
"""
```

```
    tmpfile = Path(tempfile.mktemp(suffix="_audit.log"))
```

```
    audit = open(tmpfile, "w")
```

```
def write(msg):
```

```
    timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
```

```
    audit.write(f"[{timestamp}] {msg}\n")
```

```
    audit.flush()
```

```
    write(f"Audit session started for user: {username}")
```

```
    success = False
```

```
try:
```

```
    yield write
```

```
    success = True
```

```
finally:
    write(f"Session ended — {'SUCCESS' if success else 'FAILURE'}")
    audit.close()

if success:
    dest = AUDIT_LOG_DIR / "user_create_audit.log"
    try:
        shutil.copy2(str(tmpfile), str(dest))
        log.info(f"Audit log saved to: {dest}")
    except PermissionError:
        log.warning("Could not write audit log — permission denied")
    else:
        log.warning("Audit session failed — discarding temp audit log")

    if tmpfile.exists():
        tmpfile.unlink()
```

— Functions

```
def check_root():
    """Ensure script runs as root."""
    if os.geteuid() != 0:
        raise PermissionError("This script must be run as root. Use sudo.")
```

```
def validate_username(username):
```

```
    """
```

```
    Validate username against system rules.
```

Raises:

```
    InvalidUsernameError: if username fails any validation rule
    """
    if not username:
        raise InvalidUsernameError("Username cannot be empty")

    if len(username) > 32:
        raise InvalidUsernameError(
            f"Username too long: {len(username)} chars (max 32)"
        )

    if not username[0].isalpha():
        raise InvalidUsernameError(
            f"Username must start with a letter, got: '{username[0]}'"
        )

    allowed = set("abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789_-")
    invalid_chars = [c for c in username if c not in allowed]
    if invalid_chars:
        raise InvalidUsernameError(
            f"Invalid characters in username: {invalid_chars}"
        )

    log.info(f"Username '{username}' passed validation")
    return username

def user_exists(username):
    """Return True if user exists on the system."""
    result = subprocess.run(
        ["id", username],
```

```
capture_output=True
)
return result.returncode == 0

def create_user(username, shell="/bin/bash"):
    """
    Create a new system user.

    Raises:
        UserExistsError: if user already exists
        UserCreationError: if useradd fails
    """

    log.info(f"Checking if user '{username}' exists...")

    try:
        if user_exists(username):
            raise UserExistsError(f"User '{username}' already exists")

        log.info(f"Creating user '{username}' with shell {shell}...")

        result = subprocess.run(
            ["useradd", "-m", "-s", shell, username],
            capture_output=True,
            text=True
        )

        if result.returncode != 0:
            raise UserCreationError(
                f"useradd failed for '{username}': {result.stderr.strip()}"
            )
    
```

```
)  
  
except (UserExistsError, UserCreationError):  
    raise # re-raise our custom exceptions as-is  
  
except Exception as e:  
    raise UserCreationError(f"Unexpected error during user creation: {e}")  
  
else:  
    # Only runs if no exception occurred  
    log.info(f"User '{username}' created successfully")  
    log.info(f"Home directory: /home/{username}")  
    log.info(f"Shell: {shell}")  
    return True  
  
finally:  
    # Always runs  
    log.debug(f"create_user() completed for '{username}'")  
  
# — Main ——————  
  
def main():  
    try:  
        # Root check  
        check_root()  
  
        # Argument check  
        if len(sys.argv) < 2:  
            raise ValueError(f"Usage: {sys.argv[0]} <username>")
```

```
username = sys.argv[1]

# Validate and create inside audit session
with audit_log_session(username) as audit:
    audit(f"Root check passed — running as UID {os.getuid()}")

    validated = validate_username(username)
    audit(f"Validation passed for: {validated}")

    create_user(validated)
    audit(f"User '{validated}' created successfully")

except PermissionError as e:
    log.error(f"Permission error: {e}")
    sys.exit(1)

except InvalidUsernameError as e:
    log.error(f"Invalid username: {e}")
    sys.exit(1)

except UserExistsError as e:
    log.error(f"User conflict: {e}")
    sys.exit(1)

except UserCreationError as e:
    log.error(f"Creation failed: {e}")
    sys.exit(1)

except ValueError as e:
```

```
log.error(str(e))

sys.exit(1)

except Exception as e:
    log.exception(f"Unexpected error: {e}")
    sys.exit(1)

# — Entry Point ——————
if __name__ == "__main__":
    main()
```

Explanation

Context manager with success/failure tracking

```
python
success = False

try:
    yield write
    success = True
finally:
    if success:
        shutil.copy2(str(tmpfile), str(dest))
    else:
        log.warning("Discarding temp audit log")
```

success starts as False. It only becomes True if the yield completes without raising an exception. The finally block then decides whether to keep or discard the temp file — exactly like trap EXIT in Bash but with full awareness of whether things succeeded.

Re-raising exceptions cleanly

```
python
```

```
except (UserExistsError, UserCreationError):
    raise # re-raise as-is
except Exception as e:
    raise UserCreationError(f"Unexpected error: {e}")
```

Bare `raise` re-raises the current exception unchanged. This lets us catch unexpected errors and wrap them in our custom type, while passing our own exceptions through untouched to be handled at the top level.

try/except/else/finally all four blocks

```
python
try: # attempt the operation
except: # handle specific failures
else: # runs ONLY on success — log confirmation here
finally: # ALWAYS runs — cleanup and debug logging
...  
---
```

Using all four blocks is the most expressive Python error handling pattern. `else` is where you put success-path logic that should only run if nothing went wrong — keeping it separate from the `try` block makes intent very clear.

```
## 🎨 Expected Output
```

```
---
```

```
[2024-01-15 10:30:01] [INFO] Root check passed
[2024-01-15 10:30:01] [INFO] Username 'john' passed validation
[2024-01-15 10:30:01] [INFO] Checking if user 'john' exists...
[2024-01-15 10:30:01] [INFO] Creating user 'john' with shell /bin/bash...
[2024-01-15 10:30:01] [INFO] User 'john' created successfully
[2024-01-15 10:30:01] [INFO] Home directory: /home/john
[2024-01-15 10:30:01] [INFO] Shell: /bin/bash
```

```
[2024-01-15 10:30:01] [INFO] Audit log saved to: /var/log/user_create_audit.log
```

Module 7 — Sysadmin Libraries

This is where Python becomes unstoppable for sysadmin work. These libraries give you capabilities that would take hundreds of lines of Bash to replicate — often in just a few lines of Python.

1. argparse — Professional CLI Arguments

In Bash you used \$1, \$2 and manual checks. argparse gives you a full CLI interface with help text, validation, and optional flags automatically:

```
python
```

```
#!/usr/bin/env python3
```

```
import argparse
```

```
def build_parser():
```

```
    parser = argparse.ArgumentParser(  
        description="System health monitoring tool",  
        epilog="Example: python3 monitor.py --disk 85 --services nginx sshd"  
    )
```

```
# Positional argument — required
```

```
    parser.add_argument(  
        "action",  
        choices=["check", "report", "fix"],  
        help="Action to perform"  
    )
```

```
# Optional flag with default
```

```
    parser.add_argument(  
        "--disk", "-d",
```

```
    type=int,  
    default=85,  
    metavar="PERCENT",  
    help="Disk usage threshold (default: 85)"  
)
```

```
# Optional flag — multiple values
```

```
parser.add_argument(  
    "--services", "-s",  
    nargs="+",  
    default=["nginx", "sshd", "cron"],  
    metavar="SERVICE",  
    help="Services to check"  
)
```

```
# Boolean flag — True if present
```

```
parser.add_argument(  
    "--verbose", "-v",  
    action="store_true",  
    help="Enable verbose output"  
)
```

```
# Output file
```

```
parser.add_argument(  
    "--output", "-o",  
    type=str,  
    default=None,  
    help="Save report to file"  
)
```

```
return parser

def main():
    parser = build_parser()
    args = parser.parse_args()

    print(f"Action : {args.action}")
    print(f"Threshold: {args.disk}%")
    print(f"Services : {args.services}")
    print(f"Verbose : {args.verbose}")
    print(f"Output : {args.output}")

if __name__ == "__main__":
    main()
```
Auto-generated help output:

```
$ python3 monitor.py --help
usage: monitor.py [-h] [--disk PERCENT] [--services SERVICE [SERVICE ...]]
                  [--verbose] [--output OUTPUT]
                  {check,report,fix}

```

System health monitoring tool

positional arguments:

{check,report,fix} Action to perform

options:

```
--disk PERCENT, -d  Disk usage threshold (default: 85)  
--services SERVICE  Services to check  
--verbose, -v       Enable verbose output  
--output OUTPUT, -o  Save report to file
```

2. psutil — System & Process Monitoring

psutil replaces top, free, df, ps all in pure Python:

```
python  
#!/usr/bin/env python3  
import psutil  
from datetime import datetime, timedelta
```

```
# — CPU ——————
```

```
cpu_percent = psutil.cpu_percent(interval=1)  
cpu_count  = psutil.cpu_count()  
cpu_freq   = psutil.cpu_freq()  
print(f"CPU Usage : {cpu_percent}%")  
print(f"CPU Cores : {cpu_count}")  
print(f"CPU Freq : {cpu_freq.current:.0f}MHz")
```

```
# — Memory ——————
```

```
mem = psutil.virtual_memory()  
swap = psutil.swap_memory()  
print(f"RAM Total : {mem.total // 2**30}GB")  
print(f"RAM Used : {mem.used // 2**30}GB ({mem.percent}%)")  
print(f"RAM Free : {mem.available // 2**30}GB")  
print(f"Swap Used : {swap.used // 2**20}MB ({swap.percent}%)")
```

```
# — Disk ——————  
for partition in psutil.disk_partitions():  
    try:  
        usage = psutil.disk_usage(partition.mountpoint)  
        print(f"{partition.mountpoint}:<15}"  
              f"\n{usage.total // 2**30}GB total "  
              f"\n{usage.percent}% used")  
    except PermissionError:  
        continue
```

```
# — Network ——————  
net = psutil.net_io_counters()  
print(f"Bytes Sent : {net.bytes_sent // 2**20}MB")  
print(f"Bytes Received: {net.bytes_recv // 2**20}MB")
```

```
# Network connections  
for conn in psutil.net_connections(kind="inet"):  
    if conn.status == "LISTEN":  
        print(f"Listening on port: {conn.laddr.port}")
```

```
# — Processes ——————  
# Top 5 memory-consuming processes  
processes = []  
for proc in psutil.process_iter(["pid", "name", "memory_percent", "cpu_percent"]):  
    try:  
        processes.append(proc.info)  
    except psutil.NoSuchProcess:  
        pass
```

```
top5_mem = sorted(processes, key=lambda x: x["memory_percent"], reverse=True)[:5]
print(f"\n{'PID':<8} {'NAME':<20} {'MEM%':<8} {'CPU%'})")
print("-" * 45)
for p in top5_mem:
    print(f"{p['pid']:<8} {p['name']:<20} "
          f"{p['memory_percent']:.1f}% {p['cpu_percent']:.1f}%")
```

```
# — Uptime ——————
boot_time = datetime.fromtimestamp(psutil.boot_time())
uptime = datetime.now() - boot_time
print(f"\nSystem uptime: {uptime}")
```

3. paramiko — SSH Automation

Run commands on remote servers without leaving Python — replaces complex SSH loops in Bash:

```
python
#!/usr/bin/env python3
import paramiko
import logging

log = logging.getLogger()

def create_ssh_client(hostname, username, key_path=None, password=None):
    """Create and return an SSH connection."""
    client = paramiko.SSHClient()
    client.set_missing_host_key_policy(paramiko.AutoAddPolicy())

    try:
        if key_path:
            client.connect(hostname, username=username,
```

```
key_filename=key_path, timeout=10)

else:

    client.connect(hostname, username=username,
                  password=password, timeout=10)

    log.info(f"Connected to {hostname}")

    return client


except paramiko.AuthenticationException:

    log.error(f"Authentication failed for {hostname}")

    raise

except paramiko.SSHException as e:

    log.error(f"SSH error for {hostname}: {e}")

    raise


def run_remote_command(client, command):

    """Run a command on a remote server and return output."""

    stdin, stdout, stderr = client.exec_command(command)

    output = stdout.read().decode().strip()

    error = stderr.read().decode().strip()

    exit_code = stdout.channel.recv_exit_status()

    return output, error, exit_code


def check_remote_servers(servers, key_path=".ssh/id_rsa"):

    """Check health of multiple remote servers over SSH."""

    results = {}


for hostname in servers:

    client = None

    try:
```

```
client = create_ssh_client(hostname, "admin",
                           key_path=key_path)

# Run multiple commands
commands = {
    "uptime" : "uptime -p",
    "disk"   : "df / | awk 'NR==2 {print $5}'",
    "memory" : "free -m | awk '/Mem:/ {printf \"%.\n0f%%\", $3/$2*100}'",
    "os"     : "cat /etc/os-release | grep PRETTY_NAME | cut -d= -f2",
}

server_data = {}
for key, cmd in commands.items():
    output, _, exit_code = run_remote_command(client, cmd)
    server_data[key] = output if exit_code == 0 else "ERROR"

results[hostname] = {"status": "reachable", **server_data}
log.info(f"{hostname}: OK")

except Exception as e:
    results[hostname] = {"status": "unreachable", "error": str(e)}
    log.error(f"{hostname}: FAILED — {e}")

finally:
    if client:
        client.close()

return results
```

```

# Usage

servers = ["web01", "web02", "db01"]

results = check_remote_servers(servers)

print(f"\n{'='*60}")

print(f" {'SERVER':<12} {'STATUS':<14} {'DISK':<8} {'MEM':<8} {'UPTIME'}")

print(f" {'-'*55}")

for host, data in results.items():

    if data["status"] == "reachable":

        print(f" {host:<12} {'UP':<14} "
              f"{data.get('disk','?'):<8} "
              f"{data.get('memory','?'):<8} "
              f"{data.get('uptime','?')}")

    else:

        print(f" {host:<12} {'UNREACHABLE':<14}")

```

4. schedule — Task Scheduling in Python

```

python

#!/usr/bin/env python3

import schedule

import time

import logging

import psutil

from datetime import datetime


log = logging.getLogger()

logging.basicConfig(level=logging.INFO,
                    format="[%(asctime)s] %(message)s",
                    datefmt="%Y-%m-%d %H:%M:%S")

```

```
def check_disk():

    """Check disk usage — runs every 5 minutes."""
    usage = psutil.disk_usage("/").percent
    if usage > 85:
        log.warning(f"DISK ALERT: / is at {usage}%")
    else:
        log.info(f"Disk OK: {usage}%")

def check_memory():

    """Check memory usage — runs every minute."""
    mem = psutil.virtual_memory()
    if mem.percent > 90:
        log.warning(f"MEMORY ALERT: {mem.percent}% used")
    else:
        log.info(f"Memory OK: {mem.percent}%")

def daily_report():

    """Generate daily report — runs at 7 AM."""
    log.info("Generating daily health report... ")
    # ... report generation code

# Schedule jobs

schedule.every(5).minutes.do(check_disk)
schedule.every(1).minutes.do(check_memory)
schedule.every().day.at("07:00").do(daily_report)
schedule.every().monday.at("09:00").do(daily_report)

log.info("Scheduler started — press Ctrl+C to stop")
```

```
# Run loop  
while True:  
    schedule.run_pending()  
    time.sleep(30)
```

5. `smtplib` — Sending Alert Emails

```
python  
#!/usr/bin/env python3  
import smtplib  
import logging  
from email.mime.text import MIMEText  
from email.mime.multipart import MIMEMultipart  
from datetime import datetime
```

```
log = logging.getLogger()  
  
def send_alert_email(subject, body, to_email,  
                     from_email="alerts@yourserver.com",  
                     smtp_host="localhost"):
```

"""

Send an alert email via SMTP.

Works with local sendmail or external SMTP.

"""

try:

```
    msg = MIMEMultipart()  
    msg["From"] = from_email  
    msg["To"] = to_email  
    msg["Subject"] = f"[SERVER ALERT] {subject}"
```

```
# Add timestamp to body

full_body = (
    f"Alert Time: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}\n"
    f"{'='*40}\n\n"
    f"{body}\n\n"
    f"{'='*40}\n"
    f"Sent by automated monitoring system"
)

msg.attach(MIMEText(full_body, "plain"))
```

```
with smtplib.SMTP(smtp_host, 25) as smtp:
    smtp.sendmail(from_email, to_email, msg.as_string())
```

```
log.info(f"Alert email sent to {to_email}: {subject}")
return True
```

```
except smtplib.SMTPException as e:
    log.error(f"Failed to send alert email: {e}")
    return False
```

```
# Usage

import psutil

disk = psutil.disk_usage("/").percent

if disk > 85:
    send_alert_email(
        subject=f"Disk Usage Critical: {disk}%",
        body=f"Root partition is at {disk}% usage.\nImmediate action required.",
        to_email="admin@yourcompany.com"
```

)

6. configparser — Reading Config Files

python

```
#!/usr/bin/env python3

import configparser
from pathlib import Path


# — Read an INI-style config file ——————
# Config file: /etc/mymonitor.conf

# [general]
# log_level = INFO
# log_file = /var/log/monitor.log
#
# [thresholds]
# disk_warning = 75
# disk_critical = 90
# memory_warning = 80
#
# [services]
# check = nginx,sshd,cron,rsyslog
# restart_on_failure = true


def load_config(config_path="/etc/mymonitor.conf"):
    """Load and validate configuration file."""
    config = configparser.ConfigParser()

    if not Path(config_path).exists():
        raise FileNotFoundError(f"Config file not found: {config_path}")
```

```

config.read(config_path)

# Extract with type conversion and defaults
return {
    "log_level" : config.get("general", "log_level", fallback="INFO"),
    "log_file" : config.get("general", "log_file", fallback="/var/log/monitor.log"),
    "disk_warn" : config.getint("thresholds", "disk_warning", fallback=75),
    "disk_crit" : config.getint("thresholds", "disk_critical", fallback=90),
    "mem_warn" : config.getint("thresholds", "memory_warning", fallback=80),
    "services" : config.get("services", "check", fallback="sshd,cron").split(","),
    "auto_restart": config.getboolean("services", "restart_on_failure", fallback=False),
}

# Write a config file
def write_default_config(path):
    config = configparser.ConfigParser()
    config["general"] = {"log_level": "INFO",
                         "log_file" : "/var/log/monitor.log"}
    config["thresholds"] = {"disk_warning" : "75",
                           "disk_critical": "90",
                           "memory_warning": "80"}
    config["services"] = {"check" : "nginx,sshd,cron",
                          "restart_on_failure": "true"}
    with open(path, "w") as f:
        config.write(f)
    print(f"Default config written to {path}")

```

7. Real-World Example — Complete Monitoring Tool

```
python  
#!/usr/bin/env python3  
# =====  
# Script Name: monitor.py  
# Description: Complete system monitor using  
#     argparse + psutil + logging  
# Usage   : python3 monitor.py check --disk 85 --verbose  
# =====
```

```
import sys  
import os  
import logging  
import argparse  
import psutil  
import smtplib  
from datetime import datetime  
from pathlib import Path  
from email.mime.text import MIMEText
```

```
# — Argument Parser _____  
def build_parser():  
    parser = argparse.ArgumentParser(  
        description="Linux System Monitor",  
        formatter_class=argparse.RawDescriptionHelpFormatter,  
        epilog="")
```

Examples:

```
python3 monitor.py check  
python3 monitor.py check --disk 80 --memory 85  
python3 monitor.py report --output /tmp/report.txt
```

```

python3 monitor.py check --services nginx sshd --verbose
    """
)
parser.add_argument("action",
    choices=["check", "report"],
    help="Action to perform")
parser.add_argument("--disk", type=int, default=85,
    help="Disk threshold %% (default: 85)")
parser.add_argument("--memory", type=int, default=90,
    help="Memory threshold %% (default: 90)")
parser.add_argument("--services", nargs="+",
    default=["sshd", "cron", "rsyslog"],
    help="Services to check")
parser.add_argument("--output", type=str, default=None,
    help="Save report to file")
parser.add_argument("--email", type=str, default=None,
    help="Send alerts to this email")
parser.add_argument("--verbose", action="store_true",
    help="Verbose output")
return parser

```

```

# — Logger ——————
def setup_logger(verbose=False):
    level = logging.DEBUG if verbose else logging.INFO
    logging.basicConfig(
        level=level,
        format='[%(asctime)s] [%(levelname)s] %(message)s',
        datefmt="%Y-%m-%d %H:%M:%S",
        handlers=[logging.StreamHandler(sys.stdout)]

```

```

    )

return logging.getLogger()

# — Check Functions ——————
def check_disk(threshold):
    results = []
    for part in psutil.disk_partitions():
        try:
            usage = psutil.disk_usage(part.mountpoint)
            status = "CRITICAL" if usage.percent >= threshold else \
                     "WARNING" if usage.percent >= threshold - 10 else "OK"
            results.append({
                "mount" : part.mountpoint,
                "total" : f"{usage.total // 2**30}GB",
                "used" : f"{usage.used // 2**30}GB",
                "percent": usage.percent,
                "status" : status
            })
        except PermissionError:
            continue
    return results

def check_memory(threshold):
    mem = psutil.virtual_memory()
    swap = psutil.swap_memory()
    status = "CRITICAL" if mem.percent >= threshold else \
             "WARNING" if mem.percent >= threshold - 10 else "OK"
    return {
        "ram_total" : f"{mem.total // 2**30}GB",

```

```
"ram_used" : f"{mem.used // 2**30}GB",
"ram_percent": mem.percent,
"swap_used" : f"{swap.used // 2**20}MB",
"status" : status
}

def check_services(service_list):
    import subprocess
    results = []
    for svc in service_list:
        r = subprocess.run(["systemctl", "is-active", svc],
                          capture_output=True, text=True)
        results.append({
            "service": svc,
            "status" : "RUNNING" if r.returncode == 0 else "DOWN"
        })
    return results

def check_cpu():
    cpu = psutil.cpu_percent(interval=1)
    load = os.getloadavg()
    return {
        "usage" : cpu,
        "load_1" : load[0],
        "load_5" : load[1],
        "load_15": load[2],
        "status" : "WARNING" if cpu > 80 else "OK"
    }
```

```
# — Report Functions ——————  
  
def print_report(disk, memory, services, cpu, args):  
  
    lines = []  
  
    lines.append(f"\n{'='*60}")  
  
    lines.append(f" System Health Report")  
  
    lines.append(f" {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}\")")  
  
    lines.append(f" Host: {os.uname().nodename}\")")  
  
    lines.append(f"{'='*60}")  
  
  
    lines.append(f"\n CPU")  
  
    lines.append(f" {'-'*40}")  
  
    lines.append(f" Usage : {cpu['usage']}% [{cpu['status']}]\")")  
  
    lines.append(f" Load Avg : {cpu['load_1']} / {cpu['load_5']} / {cpu['load_15']}\")")  
  
  
    lines.append(f"\n Memory")  
  
    lines.append(f" {'-'*40}")  
  
    lines.append(f" RAM : {memory['ram_used']} / {memory['ram_total']} \"  
        f"({memory['ram_percent']}%) [{memory['status']}]\")")  
  
    lines.append(f" Swap : {memory['swap_used']} used")  
  
  
    lines.append(f"\n Disk Usage")  
  
    lines.append(f" {'-'*40}")  
  
    for d in disk:  
  
        lines.append(f" {d['mount']}<12} {d['used']}>6}/{d['total']}<6} \"  
            f"({d['percent']}>6.1f)% [{d['status']}]\")")  
  
  
    lines.append(f"\n Services")  
  
    lines.append(f" {'-'*40}")  
  
    for svc in services:
```

```
icon = "✓" if svc["status"] == "RUNNING" else "✗"
lines.append(f" {icon} {svc['service']:<20} {svc['status']}")

lines.append(f"{'='*60}\n")

report_text = "\n".join(lines)

print(report_text)

if args.output:
    Path(args.output).write_text(report_text)
    print(f" Report saved to: {args.output}")

# — Main ——————
def main():
    parser = build_parser()
    args = parser.parse_args()
    log = setup_logger(args.verbose)

    log.info("===== Monitor Started =====")

    disk = check_disk(args.disk)
    memory = check_memory(args.memory)
    services = check_services(args.services)
    cpu = check_cpu()

    print_report(disk, memory, services, cpu, args)

    # Exit with error if any critical issue found
    has_critical = (
```

```

any(d["status"] == "CRITICAL" for d in disk) or
memory["status"] == "CRITICAL" or
any(s["status"] == "DOWN" for s in services)
)

if has_critical:
    log.error("Critical issues detected!")
    sys.exit(1)

log.info("===== Monitor Completed =====")

```

```

if __name__ == "__main__":
    main()

```

Key Takeaways from Module 7

- argparse gives you a professional CLI with help text, types, and defaults automatically
- psutil replaces top, free, df, ps — pure Python, no subprocess needed
- paramiko lets you SSH into remote servers and run commands from Python
- schedule gives you cron-like scheduling inside a Python script
- smtplib sends alert emails without needing mailx or sendmail from Bash
- configparser reads INI-style config files — great for making scripts configurable
- Install third-party libraries with pip3 install psutil paramiko schedule

Your Exercise — Module 7

Write a Python script called sysmon.py using argparse and psutil that:

1. Uses argparse with these arguments: --disk threshold, --memory threshold, --interval seconds, --count how many times to check, --output report file
2. Uses psutil to collect CPU, memory, and disk stats
3. Runs the check --count times with --interval seconds between each run

4. Prints a clean formatted table each run showing all metrics
 5. At the end writes a summary report to --output if specified
 6. Exits with code 1 if any threshold was breached during any run
-

Say "show solution" and we move to **Module 8 — Real-World Python Projects!** 🎉

show solution and move to next module

11:39 PM

Solution — sysmon.py

python

```
#!/usr/bin/env python3

# =====

# Script Name: sysmon.py

# Description: System monitor using argparse

#       and psutil

# Usage   : python3 sysmon.py --disk 85
#           --memory 90 --interval 5
#           --count 3 --output /tmp/report.txt

# =====
```

```
import sys
import os
import time
import logging
import argparse
import psutil
from datetime import datetime
from pathlib import Path

# — Argument Parser ——————
```

```
def build_parser():

    parser = argparse.ArgumentParser(
        description="System Resource Monitor",
        formatter_class=argparse.RawDescriptionHelpFormatter,
        epilog="")
```

Examples:

```
python3 sysmon.py

python3 sysmon.py --disk 80 --memory 85

python3 sysmon.py --interval 10 --count 6

python3 sysmon.py --count 3 --output /tmp/report.txt

""")

)
```

```
parser.add_argument("--disk", type=int, default=85,
                    metavar="PCT",
                    help="Disk usage alert threshold (default: 85%%)")

parser.add_argument("--memory", type=int, default=90,
                    metavar="PCT",
                    help="Memory usage alert threshold (default: 90%%)")

parser.add_argument("--cpu", type=int, default=80,
                    metavar="PCT",
                    help="CPU usage alert threshold (default: 80%%)")

parser.add_argument("--interval", type=int, default=5,
                    metavar="SEC",
                    help="Seconds between checks (default: 5)")

parser.add_argument("--count", type=int, default=3,
                    metavar="N",
                    help="Number of checks to run (default: 3)")

parser.add_argument("--output", type=str, default=None,
                    metavar="FILE",
```

```
    help="Save summary report to this file")
parser.add_argument("--verbose", action="store_true",
                   help="Show extra detail per check")
return parser
```

```
# — Logger ——————
def setup_logger(verbose=False):
    level = logging.DEBUG if verbose else logging.INFO
    logging.basicConfig(
        level=level,
        format="[%(asctime)s] [%(levelname)s] %(message)s",
        datefmt="%Y-%m-%d %H:%M:%S",
        handlers=[logging.StreamHandler(sys.stdout)])
)
return logging.getLogger()
```

```
# — Collection Functions ——————
```

```
def collect_cpu(threshold):
    """Collect CPU metrics using psutil."""
    usage = psutil.cpu_percent(interval=1)
    load = os.getloadavg()
    return {
        "usage" : usage,
        "load_1" : round(load[0], 2),
        "load_5" : round(load[1], 2),
        "load_15" : round(load[2], 2),
        "status" : "CRITICAL" if usage >= threshold else
                   "WARNING" if usage >= threshold - 10 else "OK"
    }
```

```
def collect_memory(threshold):
    """Collect memory metrics using psutil."""
    mem = psutil.virtual_memory()
    swap = psutil.swap_memory()
    return {
        "total_gb" : round(mem.total / 2**30, 1),
        "used_gb" : round(mem.used / 2**30, 1),
        "percent" : mem.percent,
        "swap_used" : round(swap.used / 2**20, 1),
        "swap_total" : round(swap.total / 2**20, 1),
        "status" : "CRITICAL" if mem.percent >= threshold else
                    "WARNING" if mem.percent >= threshold - 10 else "OK"
    }
```

```
def collect_disk(threshold):
    """Collect disk metrics for all partitions using psutil."""
    results = []
    for part in psutil.disk_partitions():
        try:
            usage = psutil.disk_usage(part.mountpoint)
            results.append({
                "mount" : part.mountpoint,
                "total_gb": round(usage.total / 2**30, 1),
                "used_gb" : round(usage.used / 2**30, 1),
                "free_gb" : round(usage.free / 2**30, 1),
                "percent" : usage.percent,
                "status" : "CRITICAL" if usage.percent >= threshold else
                            "WARNING" if usage.percent >= threshold - 10 else "OK"
            })
        except:
            pass
```

```

        })

    except PermissionError:

        continue

    return results

def collect_snapshot(args):
    """Collect a full system snapshot and return as dict."""
    return {
        "timestamp": datetime.now().strftime("%Y-%m-%d %H:%M:%S"),
        "cpu"      : collect_cpu(args.cpu),
        "memory"   : collect_memory(args.memory),
        "disk"     : collect_disk(args.disk),
    }

```

— Display Functions ——————

```

def print_snapshot(snap, run_num, total_runs):
    """Print a formatted snapshot table."""

    cpu = snap["cpu"]
    mem = snap["memory"]

    print(f"\n{'='*60}")
    print(f" Check {run_num}/{total_runs} — {snap['timestamp']}")
    print(f"{'='*60}")

    # CPU row
    print(f" {'RESOURCE':<14} {'VALUE':>10} {'STATUS'}")
    print(f" {'-'*50}")

    print(f" {'CPU Usage':<14} {str(cpu['usage'])+'%':>10} {cpu['status']}")
    print(f" {'Load (1/5/15)':<14} "

```

```

f"{{str(cpu['load_1'])+'/' +str(cpu['load_5'])+'/' +str(cpu['load_15'])):>10}}"

# Memory row

mem_str = f"{{mem['used_gb']}}GB/{{mem['total_gb']}}GB"
print(f" {RAM Usage:<14} {str(mem['percent']):%'>10} {mem['status']}")

print(f" {RAM Detail:<14} {mem_str:>10}")

print(f" {Swap Used:<14} {str(mem['swap_used'])+'MB':>10}")


# Disk rows

print(f" {'-*50}")

for d in snap["disk"]:

    label = f"Disk {d['mount']}"
    val = f"{{d['percent']}%}"
    detail = f"{{d['used_gb']}}GB/{{d['total_gb']}}GB"
    print(f" {label:<14} {val:>10} {d['status']}")

    print(f" {"<14} {detail:>10}")



print(f"{'=*60}')


# — Summary Report _____

def write_summary(snapshots, args):
    """Write a summary report from all collected snapshots."""
    lines = []
    lines.append("=" * 60)
    lines.append(" SYSMON — Summary Report")
    lines.append(f" Generated : {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}")
    lines.append(f" Host     : {os.uname().nodename}")
    lines.append(f" Checks   : {len(snapshots)}")
    lines.append(f" Interval : {args.interval}s")

```

```

lines.append("= * 60)

# CPU summary
cpu_values = [s["cpu"]["usage"] for s in snapshots]
lines.append(f"\n CPU Summary")
lines.append(f" {'-'*40}")

lines.append(f" Average : {round(sum(cpu_values)/len(cpu_values), 1)}%")
lines.append(f" Peak   : {max(cpu_values)}%")
lines.append(f" Minimum : {min(cpu_values)}%")

# Memory summary
mem_values = [s["memory"]["percent"] for s in snapshots]
lines.append(f"\n Memory Summary")
lines.append(f" {'-'*40}")

lines.append(f" Average : {round(sum(mem_values)/len(mem_values), 1)}%")
lines.append(f" Peak   : {max(mem_values)}%")
lines.append(f" Minimum : {min(mem_values)}%")

# Threshold breaches
breaches = []
for i, snap in enumerate(snapshots, 1):
    if snap["cpu"]["status"] != "OK":
        breaches.append(f" Run {i}: CPU at {snap['cpu']['usage']}%")
    if snap["memory"]["status"] != "OK":
        breaches.append(f" Run {i}: Memory at {snap['memory']['percent']}%")
    for d in snap["disk"]:
        if d["status"] != "OK":
            breaches.append(f" Run {i}: Disk {d['mount']} at {d['percent']}%")

```

```
lines.append(f"\n Threshold Breaches: {len(breaches)}")  
lines.append(f" {'-'*40}")  
  
if breaches:  
    lines.extend(breaches)  
  
else:  
    lines.append(" None — all resources within thresholds")  
  
  
lines.append(f"\n{'='*60}\n")  
report = "\n".join(lines)  
  
  
if args.output:  
    Path(args.output).write_text(report)  
    print(f"\n Report saved to: {args.output}")  
  
  
return report
```

```
# — Breach Detection ——————  
  
def has_breach(snapshots):  
    """Return True if any snapshot has a non-OK status."""  
  
    for snap in snapshots:  
        if snap["cpu"]["status"] != "OK":  
            return True  
  
        if snap["memory"]["status"] != "OK":  
            return True  
  
        if any(d["status"] != "OK" for d in snap["disk"]):  
            return True  
  
    return False
```

```
# — Main ——————
```

```
def main():

    parser = build_parser()

    args = parser.parse_args()

    log = setup_logger(args.verbose)

    snapshots = []

    log.info(f"Starting sysmon — {args.count} check(s) "
            f"every {args.interval}s")

    log.info(f"Thresholds — CPU: {args.cpu}% "
            f"Memory: {args.memory}% Disk: {args.disk}%")

    for run in range(1, args.count + 1):

        try:
            snap = collect_snapshot(args)
            snapshots.append(snap)
            print_snapshot(snap, run, args.count)

            if run < args.count:
                log.debug(f"Waiting {args.interval}s before next check...")
                time.sleep(args.interval)

        except KeyboardInterrupt:
            log.warning("Interrupted by user")
            break

    except Exception as e:
        log.exception(f"Error during check {run}: {e}")
        continue
```

```
# Write summary

if snapshots:

    print("\n")
    write_summary(snapshots, args)

# Exit with error if any threshold breached

if has_breach(snapshots):

    log.error("One or more thresholds were breached!")
    sys.exit(1)

log.info("All checks completed — no breaches detected")

# — Entry Point ——————
```

```
if __name__ == "__main__":
    main()
```

Explanation

argparse with metavar for clean help text

python

```
parser.add_argument("--disk", type=int, default=85,
                    metavar="PCT",
                    help="Disk usage alert threshold (default: 85%)")
```

metavar="PCT" replaces the default DISK placeholder in help output with the more descriptive PCT. The % in the help string escapes the % so it displays correctly in the auto-generated help text.

os.getloadavg() for load averages

python

```
load = os.getloadavg() # returns (1min, 5min, 15min) tuple
```

This is a clean Python alternative to parsing uptime output. Returns a tuple of three floats you unpack directly — no subprocess or string parsing needed.

Average, peak, and minimum from a list

python

```
cpu_values = [s["cpu"]["usage"] for s in snapshots]
avg = round(sum(cpu_values) / len(cpu_values), 1)
peak = max(cpu_values)
minimum = min(cpu_values)
---
```

List comprehension collects all CPU values across runs. Python's built-in `sum()`, `max()`, `min()` then give you stats in three clean lines — no loops needed.

📈 Expected Output

```
[2024-01-15 10:30:01] [INFO] Starting sysmon — 3 checks every 5s
```

```
[2024-01-15 10:30:01] [INFO] Thresholds — CPU: 80% Memory: 90% Disk: 85%
```

```
=====
```

```
Check 1/3 — 2024-01-15 10:30:02
```

```
=====
```

RESOURCE	VALUE	STATUS
----------	-------	--------

```
-----
```

CPU Usage	12.4%	OK
-----------	-------	----

Load (1/5/15)	0.5/0.4/0.3	
---------------	-------------	--

RAM Usage	67.2%	OK
-----------	-------	----

RAM Detail	10.7GB/16.0GB	
------------	---------------	--

```
Swap Used      0.0MB
```

```
Disk /      43.2%  OK
```

```
Disk /home   21.0%  OK
```

SYSMON — Summary Report

Generated : 2024-01-15 10:30:17

Host : web-server-01

Checks : 3

CPU Summary

Average : 13.1% Peak: 15.2% Minimum: 12.4%

Memory Summary

Average : 67.5% Peak: 68.1% Minimum: 67.2%

Threshold Breaches: 0 — all resources within thresholds

Module 8 — Real-World Python Projects

Everything comes together here. Five complete production-grade Python scripts that you can deploy and use immediately as a Linux administrator.

Project 1 — System Health Monitor (`health_monitor.py`)

```
python  
#!/usr/bin/env python3  
# ======  
# Script Name: health_monitor.py  
# Description: Full system health monitor with  
#               psutil, logging, email alerts  
# Usage     : sudo python3 health_monitor.py
```

```
# Schedule : */5 * * * * python3 /opt/scripts/health_monitor.py
# =====
```

```
import os
import sys
import logging
import psutil
import subprocess
import smtplib
from datetime import datetime
from pathlib import Path
from email.mime.text import MIMEText
```

```
# — Config ——————
```

```
LOGFILE      = "/var/log/health_monitor.log"
REPORT_DIR    = Path("/tmp/health_reports")
ALERT_EMAIL   = "admin@yourcompany.com"
DISK_CRITICAL = 90
DISK_WARNING   = 75
MEMORY_CRITICAL = 90
MEMORY_WARNING = 80
CPU_WARNING    = 85
SERVICES      = ["sshd", "cron", "rsyslog"]
```

```
# — Logger ——————
```

```
logging.basicConfig(
    level=logging.INFO,
    format="[%(asctime)s] [%(levelname)s] %(message)s",
    datefmt="%Y-%m-%d %H:%M:%S",
```

```

handlers=[

    logging.StreamHandler(sys.stdout),

    logging.FileHandler(LOGFILE)

]

)

log = logging.getLogger()

# — Alert ——————
def send_alert(subject, body):

    try:

        msg = MIMEText(body)

        msg["Subject"] = f"[ALERT] {subject}"

        msg["From"] = "monitor@localhost"

        msg["To"] = ALERT_EMAIL

        with smtplib.SMTP("localhost", 25) as s:

            s.sendmail("monitor@localhost", ALERT_EMAIL, msg.as_string())

            log.info(f"Alert sent: {subject}")

    except Exception as e:

        log.warning(f"Could not send alert email: {e}")

# — Check Functions ——————
def check_cpu():

    usage = psutil.cpu_percent(interval=2)

    load = os.getloadavg()

    if usage >= CPU_WARNING:

        log.warning(f"CPU high: {usage}% | Load: {load[0]}/{load[1]}/{load[2]}")

        send_alert(f"CPU High: {usage}%",

                  f"CPU usage is {usage}%\nLoad average: {load}")

    else:

```

```

log.info(f"CPU OK: {usage}% | Load: {load[0]:.2f}/{load[1]:.2f}/{load[2]:.2f}")

return usage


def check_memory():

    mem = psutil.virtual_memory()

    if mem.percent >= MEMORY_CRITICAL:

        log.error(f"Memory CRITICAL: {mem.percent}%")
        send_alert("Memory Critical", f"Memory at {mem.percent}%")

    elif mem.percent >= MEMORY_WARNING:

        log.warning(f"Memory WARNING: {mem.percent}%")

    else:

        log.info(f"Memory OK: {mem.percent}% used")

    return mem.percent


def check_disk():

    issues = []

    for part in psutil.disk_partitions():

        try:

            usage = psutil.disk_usage(part.mountpoint)

            pct = usage.percent

            if pct >= DISK_CRITICAL:

                log.error(f"Disk CRITICAL: {part.mountpoint} at {pct}%")
                issues.append(f"{part.mountpoint} at {pct}%")
                send_alert(f"Disk Critical: {part.mountpoint}",
                          f"Disk {part.mountpoint} is at {pct}%")

            elif pct >= DISK_WARNING:

                log.warning(f"Disk WARNING: {part.mountpoint} at {pct}%")

            else:

                log.info(f"Disk OK: {part.mountpoint} at {pct}%")

        except Exception as e:
            log.error(f"Error checking disk {part.mountpoint}: {e}")

```

```
except PermissionError:
    continue
return issues

def check_services():
    down = []
    for svc in SERVICES:
        result = subprocess.run(["systemctl", "is-active", svc],
                               capture_output=True, text=True)
        if result.returncode == 0:
            log.info(f"Service OK: {svc}")
        else:
            log.error(f"Service DOWN: {svc} — attempting restart")
            restart = subprocess.run(["systemctl", "restart", svc],
                                    capture_output=True)
            if restart.returncode == 0:
                log.info(f"Service {svc} restarted successfully")
            else:
                log.error(f"Service {svc} FAILED to restart!")
                down.append(svc)
    return down

def check_zombies():
    zombies = [p for p in psutil.process_iter(["pid", "status", "name"])
               if p.info["status"] == psutil.STATUS_ZOMBIE]
    if zombies:
        log.warning(f"Found {len(zombies)} zombie process(es)")
    else:
        log.info("No zombie processes found")
```

```

return zombies

def generate_report(cpu, memory, disk_issues, svc_down, zombies):
    REPORT_DIR.mkdir(parents=True, exist_ok=True)
    report_file = REPORT_DIR / f"health_{datetime.now().strftime('%Y%m%d_%H%M%S')}.txt"
    lines = [
        "=" * 50,
        f" Health Report — {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}",
        f" Host: {os.uname().nodename}",
        "=" * 50,
        f" CPU Usage : {cpu}%",
        f" Memory Usage : {memory}%",
        f" Disk Issues : {len(disk_issues)}",
        f" Services Down: {len(svc_down)}",
        f" Zombies : {len(zombies)}",
        "=" * 50,
    ]
    report_file.write_text("\n".join(lines))
    log.info(f"Report saved: {report_file}")

```

```

# — Main ——————
def main():

    log.info("===== Health Monitor Started =====")
    cpu      = check_cpu()
    memory   = check_memory()
    disk_iss = check_disk()
    svc_down = check_services()
    zombies  = check_zombies()

    generate_report(cpu, memory, disk_iss, svc_down, zombies)

```

```
log.info("===== Health Monitor Completed =====")  
if disk_iss or svc_down:  
    sys.exit(1)  
  
if __name__ == "__main__":  
    main()
```

Project 2 — Automated Backup (auto_backup.py)

```
python  
#!/usr/bin/env python3  
# ======  
# Script Name: auto_backup.py  
# Description: Automated backup with retention,  
#               integrity check, and reporting  
# Usage   : sudo python3 auto_backup.py <source>  
# Schedule : 0 2 * * * python3 /opt/scripts/auto_backup.py /var/www  
# ======  
  
import os  
import sys  
import shutil  
import hashlib  
import logging  
import tarfile  
import tempfile  
from pathlib import Path  
from datetime import datetime  
from contextlib import contextmanager
```

```
BACKUP_ROOT  = Path("/backup")
LOGFILE      = "/var/log/auto_backup.log"
RETENTION_DAYS = 7

logging.basicConfig(
    level=logging.INFO,
    format="[%(asctime)s] [%(levelname)s] %(message)s",
    datefmt="%Y-%m-%d %H:%M:%S",
    handlers=[
        logging.StreamHandler(sys.stdout),
        logging.FileHandler(LOGFILE)
    ]
)
log = logging.getLogger()

class BackupError(Exception): pass
class DiskSpaceError(BackupError): pass

@contextmanager
def temp_archive():
    """Create a temp file and guarantee cleanup."""
    tmp = Path(tempfile.mktemp(suffix=".tar.gz"))
    try:
        yield tmp
    finally:
        if tmp.exists():
            tmp.unlink()
        log.debug(f"Cleaned up temp: {tmp}")
```

```
def check_root():
    if os.geteuid() != 0:
        raise PermissionError("Must run as root")

def validate_source(path_str):
    path = Path(path_str)
    if not path.exists():
        raise BackupError(f"Source not found: {path}")
    if not path.is_dir():
        raise BackupError(f"Source is not a directory: {path}")
    return path

def check_space(source, dest_dir):
    source_size = sum(f.stat().st_size for f in source.rglob("*") if f.is_file())
    _, _, free = shutil.disk_usage(str(dest_dir.parent))
    if source_size >= free:
        raise DiskSpaceError(
            f"Insufficient space. Need {source_size//2**20}MB, "
            f"free {free//2**20}MB"
        )

def compute_checksum(filepath):
    """Compute MD5 checksum of a file."""
    md5 = hashlib.md5()
    with open(filepath, "rb") as f:
        for chunk in iter(lambda: f.read(8192), b ""):
            md5.update(chunk)
    return md5.hexdigest()
```

```
def create_backup(source, dest_dir):
    dest_dir.mkdir(parents=True, exist_ok=True)
    ts      = datetime.now().strftime("%Y%m%d_%H%M%S")
    dest_file = dest_dir / f"{source.name}_{ts}.tar.gz"

    with temp_archive() as tmp:
        log.info(f"Creating archive: {source}")
        with tarfile.open(tmp, "w:gz") as tar:
            tar.add(source, arcname=source.name)

        checksum = compute_checksum(tmp)
        shutil.move(str(tmp), str(dest_file))

    checksum_file = dest_file.with_suffix(".md5")
    checksum_file.write_text(f"{checksum} {dest_file.name}\n")

    verify = compute_checksum(dest_file)
    if verify != checksum:
        raise BackupError("Integrity check FAILED — checksums do not match!")

    size = dest_file.stat().st_size // 2**20
    log.info(f"Backup complete: {dest_file} ({size}MB)")
    log.info(f"Checksum OK: {checksum}")

    return dest_file

def apply_retention(dest_dir):
    import time
    cutoff = time.time() - (RETENTION_DAYS * 86400)
    removed = 0
```

```
for f in dest_dir.glob("*.tar.gz"):  
    if f.stat().st_mtime < cutoff:  
        f.unlink()  
        f.with_suffix(".md5").unlink(missing_ok=True)  
        log.info(f"Removed old backup: {f.name}")  
        removed += 1  
  
log.info(f"Retention: removed {removed} old backup(s)")  
  
  
def main():  
    try:  
        check_root()  
        if len(sys.argv) < 2:  
            raise ValueError(f"Usage: {sys.argv[0]} <source_directory>")  
        source = validate_source(sys.argv[1])  
        dest_dir = BACKUP_ROOT / source.name  
        log.info("===== Backup Started =====")  
        check_space(source, dest_dir)  
        create_backup(source, dest_dir)  
        apply_retention(dest_dir)  
        log.info("===== Backup Completed Successfully =====")  
    except PermissionError as e:  
        log.error(f"Permission: {e}"); sys.exit(1)  
    except DiskSpaceError as e:  
        log.error(f"Disk space: {e}"); sys.exit(1)  
    except BackupError as e:  
        log.error(f"Backup error: {e}"); sys.exit(1)  
    except Exception as e:  
        log.exception(f"Unexpected: {e}"); sys.exit(1)
```

```
if __name__ == "__main__":
    main()
```

Project 3 — Log Analyzer (log_analyzer_pro.py)

```
python
#!/usr/bin/env python3
# =====
# Script Name: log_analyzer_pro.py
# Description: Analyze system logs for
#           security threats and errors
# Usage   : python3 log_analyzer_pro.py
#           --log /var/log/auth.log
#           --output /tmp/security_report.txt
# =====
```

```
import re
import sys
import argparse
import logging
from pathlib import Path
from datetime import datetime
from collections import Counter, defaultdict

logging.basicConfig(level=logging.INFO,
                    format="[%(asctime)s] [%(levelname)s] %(message)s",
                    datefmt="%Y-%m-%d %H:%M:%S")
log = logging.getLogger()

def build_parser():
```

```
p = argparse.ArgumentParser(description="Security Log Analyzer")

p.add_argument("--log",    default="/var/log/auth.log",
               help="Log file to analyze")

p.add_argument("--threshold", type=int, default=10,
               help="Failed attempt threshold for brute force detection")

p.add_argument("--output",   default=None,
               help="Save report to file")

p.add_argument("--top",     type=int, default=10,
               help="Show top N results (default 10)")

return p
```

```
def parse_auth_log(filepath, threshold):
    """Parse auth.log and extract security events."""
    ip_pattern = re.compile(r"\b(\d{1,3}(?:\.\d{1,3}){3})\b")
    user_pattern = re.compile(r"(?:for|user)\s+(\w+)", re.IGNORECASE)
```

```
failed_ips = Counter()
success_ips = Counter()
failed_users = Counter()
invalid_users = Counter()
events      = defaultdict(list)
```

```
try:
    with open(filepath, "r", errors="replace") as f:
        for line in f:
            ips = ip_pattern.findall(line)
```

```
        if "Failed password" in line:
            for ip in ips:
```

```
    failed_ips[ip] += 1

    m = user_pattern.search(line)

    if m:

        failed_users[m.group(1)] += 1

elif "Accepted password" in line or "Accepted publickey" in line:

    for ip in ips:

        success_ips[ip] += 1

elif "Invalid user" in line:

    m = user_pattern.search(line)

    if m:

        invalid_users[m.group(1)] += 1

except FileNotFoundError:

    log.error(f"Log file not found: {filepath}")

    sys.exit(1)

except PermissionError:

    log.error(f"Permission denied: {filepath} — run as root")

    sys.exit(1)

# Flag brute force IPs

brute_force = {ip: count for ip, count in failed_ips.items()

               if count >= threshold}

return {

    "failed_ips" : failed_ips,

    "success_ips" : success_ips,

    "failed_users" : failed_users,
```

```

    "invalid_users": invalid_users,
    "brute_force" : brute_force,
}

def format_report(data, args):
    """Format analysis results into a readable report."""
    lines = []
    lines.append("=" * 60)
    lines.append(" Security Log Analysis Report")
    lines.append(f" Generated : {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}")
    lines.append(f" Log file : {args.log}")
    lines.append(f" BF threshold: {args.threshold} attempts")
    lines.append("=" * 60)

    # Brute force alerts
    if data["brute_force"]:
        lines.append(f"\n 🚨 BRUTE FORCE DETECTED — {len(data['brute_force'])} IP(s)")
        lines.append(f" {'-*50}'")
        for ip, count in sorted(data["brute_force"].items(),
                               key=lambda x: x[1], reverse=True):
            lines.append(f" {ip}<20} {count:>6} attempts ⚠ ")
    else:
        lines.append("\n ✓ No brute force attempts detected")

    # Top failed IPs
    lines.append(f"\n Top {args.top} IPs — Failed Logins")
    lines.append(f" {'-*50}'")
    lines.append(f" {'IP ADDRESS':<22} {'ATTEMPTS':>10}'")
    for ip, count in data["failed_ips"].most_common(args.top):

```

```
lines.append(f" {ip:<22} {count:>10}")

# Successful logins
lines.append(f"\n  Successful Login Sources")
lines.append(f" {'-*50}'")
if data["success_ips"]:
    for ip, count in data["success_ips"].most_common(args.top):
        lines.append(f" {ip:<22} {count:>6} login(s)")

else:
    lines.append("  No successful logins found")

# Invalid usernames
lines.append(f"\n  Top Invalid Usernames Attempted")
lines.append(f" {'-*50}'")
if data["invalid_users"]:
    for user, count in data["invalid_users"].most_common(args.top):
        lines.append(f" {user:<22} {count:>6} attempt(s)")

else:
    lines.append("  None found")

lines.append(f"\n{'=*60}\n")
return "\n".join(lines)

def main():
    args = build_parser().parse_args()
    log.info(f"Analyzing: {args.log}")
    data = parse_auth_log(args.log, args.threshold)
    report = format_report(data, args)
    print(report)
```

```
if args.output:  
    Path(args.output).write_text(report)  
    log.info(f"Report saved to: {args.output}")  
  
if data["brute_force"]:  
    sys.exit(1)  
  
if __name__ == "__main__":  
    main()
```

Project 4 — User Audit Tool (user_audit.py)

```
python  
#!/usr/bin/env python3  
# ======  
# Script Name: user_audit.py  
# Description: Full system user audit —  
#       accounts, home dirs, sudo access  
# Usage   : sudo python3 user_audit.py  
#       --output /tmp/user_audit.txt  
# ======
```

```
import os  
import sys  
import subprocess  
import argparse  
import logging  
from pathlib import Path  
from datetime import datetime
```

```
logging.basicConfig(level=logging.INFO,
                    format="[%{asctime}s] [%{levelname}s] %{message}s",
                    datefmt="%Y-%m-%d %H:%M:%S")

log = logging.getLogger()

def build_parser():
    p = argparse.ArgumentParser(description="System User Audit Tool")
    p.add_argument("--output", default=None, help="Save report to file")
    p.add_argument("--shell", default="/bin/bash",
                  help="Filter by shell (default: /bin/bash)")
    p.add_argument("--verbose", action="store_true")
    return p

def get_passwd_users(shell_filter):
    """Read /etc/passwd and return list of user dicts."""
    users = []
    with open("/etc/passwd", "r") as f:
        for line in f:
            parts = line.strip().split(":")
            if len(parts) < 7:
                continue
            username, _, uid, gid, comment, home, shell = parts
            if shell == shell_filter:
                users.append({
                    "username": username,
                    "uid": int(uid),
                    "gid": int(gid),
                    "home": home,
                })
    return users
```

```
"shell" : shell,
})

return users

def get_sudo_users():
    """Return set of users with sudo access."""
    sudo_users = set()

    try:
        with open("/etc/sudoers", "r") as f:
            for line in f:
                line = line.strip()
                if line and not line.startswith("#"):
                    parts = line.split()
                    if parts and not parts[0].startswith("%"):
                        sudo_users.add(parts[0])
    except PermissionError:
        log.warning("Cannot read /etc/sudoers — run as root for full audit")

# Check sudoers.d directory
sudoers_d = Path("/etc/sudoers.d")
if sudoers_d.exists():
    for sudoers_file in sudoers_d.iterdir():
        try:
            for line in sudoers_file.read_text().splitlines():
                line = line.strip()
                if line and not line.startswith("#"):
                    parts = line.split()
                    if parts:
                        sudo_users.add(parts[0])
        except:
            log.error(f"Failed to read {sudoers_file}: {e}")
```

```
except Exception:
    continue

return sudo_users

def check_home_directory(home_path):
    """Check home directory status and return dict."""
    path = Path(home_path)
    if not path.exists():
        return {"exists": False, "files": 0, "size_mb": 0}

    try:
        files = list(path.rglob("*"))
        size = sum(f.stat().st_size for f in files if f.is_file())
        return {
            "exists": True,
            "files": len(files),
            "size_mb": round(size / 2**20, 1)
        }
    except PermissionError:
        return {"exists": True, "files": -1, "size_mb": -1}

def get_last_login(username):
    """Get last login time for a user."""
    try:
        result = subprocess.run(
            ["lastlog", "-u", username],
            capture_output=True, text=True
        )
    
```

```

lines = result.stdout.strip().split("\n")

if len(lines) > 1:

    parts = lines[1].split()

    if "Never" in lines[1]:

        return "Never logged in"

    return " ".join(parts[3:7]) if len(parts) >= 7 else "Unknown"

except Exception:

    return "Unknown"

return "Unknown"


def audit_users(args):

    """Run full user audit and return results."""

    log.info("Starting user audit...")

    users    = get_passwd_users(args.shell)

    sudo_users = get_sudo_users()

    results  = []

    for user in users:

        home_info = check_home_directory(user["home"])

        last_login = get_last_login(user["username"])

        has_sudo  = user["username"] in sudo_users or "ALL" in sudo_users

        results.append({

            **user,

            "home_exists" : home_info["exists"],

            "home_files"  : home_info["files"],

            "home_size_mb": home_info["size_mb"],

            "last_login"  : last_login,

            "has_sudo"    : has_sudo,

```

```

        })

    if args.verbose:
        log.info(f"Audited: {user['username']}")

    return results

def format_report(results, args):
    """Format audit results into report."""
    lines = []
    lines.append("=" * 70)
    lines.append(" System User Audit Report")
    lines.append(f" Generated : {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}")
    lines.append(f" Shell    : {args.shell}")
    lines.append("=" * 70)
    lines.append(f"\n {'USER':<15} {'UID':<6} {'HOME EXISTS':<13} "
                f"{'FILES':<8} {'SIZE':>8} {'SUDO':<6} LAST LOGIN")
    lines.append(f" {'-'*65}")

    sudo_users = [r for r in results if r["has_sudo"]]
    no_home   = [r for r in results if not r["home_exists"]]
    never_login = [r for r in results if r["last_login"] == "Never logged in"]

    for r in sorted(results, key=lambda x: x["uid"]):
        sudo_flag = "YES" if r["has_sudo"] else "no"
        home_flag = "YES" if r["home_exists"] else "MISSING"
        size_str  = f"{r['home_size_mb']}MB" if r["home_size_mb"] >= 0 else "?"
        files_str = str(r["home_files"]) if r["home_files"] >= 0 else "?"

        lines.append(

```

```

f" {r['username']:<15} {r['uid']:<6} {home_flag:<13} "
f" {files_str:<8} {size_str:>8} {sudo_flag:<6} {r['last_login']}"

)

lines.append(f"\n {'-*65}'")
lines.append(f" Total users : {len(results)}")
lines.append(f" Sudo access : {len(sudo_users)} — "
f"{'', '.join(u['username'] for u in sudo_users) or 'none'}")

lines.append(f" Missing homes : {len(no_home)} — "
f"{'', '.join(u['username'] for u in no_home) or 'none'}")

lines.append(f" Never logged in: {len(never_login)} — "
f"{'', '.join(u['username'] for u in never_login) or 'none'}")

lines.append(f" {'-*65}\n")
return "\n".join(lines)

def main():
    args = build_parser().parse_args()
    results = audit_users(args)
    report = format_report(results, args)
    print(report)

    if args.output:
        Path(args.output).write_text(report)
        log.info(f"Report saved: {args.output}")

if __name__ == "__main__":
    main()

```

Project 5 — Server Inventory (server_inventory.py)

```
python
#!/usr/bin/env python3
# =====
# Script Name: server_inventory.py
# Description: Full server inventory report
#       using psutil and subprocess
# Usage   : sudo python3 server_inventory.py
#           --output /tmp/inventory.txt
# =====

import os
import sys
import platform
import subprocess
import argparse
import logging
import psutil
from pathlib import Path
from datetime import datetime, timedelta

logging.basicConfig(level=logging.INFO,
                    format="[%(asctime)s] [%(levelname)s] %(message)s",
                    datefmt="%Y-%m-%d %H:%M:%S")
log = logging.getLogger()

def build_parser():
    p = argparse.ArgumentParser(description="Server Inventory Tool")
    p.add_argument("--output", default=None, help="Save to file")
    p.add_argument("--verbose", action="store_true")
```

```
return p

def run_cmd(cmd):
    """Run shell command and return stdout string."""
    try:
        r = subprocess.run(cmd, capture_output=True, text=True, timeout=5)
        return r.stdout.strip() if r.returncode == 0 else "N/A"
    except Exception:
        return "N/A"

def collect_system():
    boot = datetime.fromtimestamp(psutil.boot_time())
    uptime = datetime.now() - boot
    return {
        "Hostname" : platform.node(),
        "OS" : platform.platform(),
        "Kernel" : platform.release(),
        "Architecture" : platform.machine(),
        "Python" : platform.python_version(),
        "Uptime" : str(timedelta(seconds=int(uptime.total_seconds()))),
        "Last Boot" : boot.strftime("%Y-%m-%d %H:%M:%S"),
    }

def collect_cpu():
    freq = psutil.cpu_freq()
    return {
        "Physical Cores" : psutil.cpu_count(logical=False),
        "Logical CPUs" : psutil.cpu_count(logical=True),
        "Current Freq" : f"{freq.current:.0f}MHz" if freq else "N/A",
    }
```

```
"Max Freq"      : f"{{freq.max:.0f}MHz"    if freq else "N/A",
"Usage (1s avg)" : f"{{psutil.cpu_percent(interval=1)}%}",
"Load Avg"      : "/".join(str(round(x, 2))
                           for x in os.getloadavg()),
}
```

```
def collect_memory():

    mem = psutil.virtual_memory()

    swap = psutil.swap_memory()

    return {
        "RAM Total"  : f"{{mem.total // 2**30}GB",
        "RAM Used"   : f"{{mem.used // 2**30}GB ({mem.percent})}",
        "RAM Free"    : f"{{mem.available // 2**30}GB",
        "Swap Total"  : f"{{swap.total // 2**20}MB",
        "Swap Used"   : f"{{swap.used // 2**20}MB ({swap.percent})}",
    }
```

```
def collect_disk():

    rows = []

    for part in psutil.disk_partitions():

        try:
            u = psutil.disk_usage(part.mountpoint)
            rows.append((part.device, part.mountpoint,
                         f"{{u.total//2**30}GB",
                         f"{{u.used //2**30}GB",
                         f"{{u.percent}}"))
        except PermissionError:
            continue

    return rows
```

```

def collect_network():

    info = {}

    addrs = psutil.net_if_addrs()
    stats = psutil.net_if_stats()

    for iface, addr_list in addrs.items():

        for addr in addr_list:

            if addr.family.name == "AF_INET":

                up = stats[iface].isup if iface in stats else False

                info[iface] = {

                    "IP" : addr.address,
                    "Netmask": addr.netmask,
                    "Status" : "UP" if up else "DOWN"
                }

    io = psutil.net_io_counters()

    info["_totals"] = {

        "Bytes Sent": f"{io.bytes_sent//2**20}MB",
        "Bytes Recv": f"{io.bytes_recv//2**20}MB",
    }

    return info

```

```

def collect_processes():

    procs = []

    for p in psutil.process_iter(["pid","name","username",
        "memory_percent","cpu_percent"]):

        try:

            procs.append(p.info)

        except psutil.NoSuchProcess:

            pass

```

```

return sorted(procs, key=lambda x: x["memory_percent"], reverse=True)[:10]

def format_section(title, data):
    lines = [f"\n {'='*55}", f" {title}", f" {'-'*55}"]
    if isinstance(data, dict):
        for k, v in data.items():
            lines.append(f" {k:<22}: {v}")
    elif isinstance(data, list):
        if data and isinstance(data[0], tuple):
            lines.append(f" {'DEVICE':<15} {'MOUNT':<12} "
                        f"{'TOTAL':>8} {'USED':>8} {'USE%':>6}")
            lines.append(f" {'-'*52}")
        for row in data:
            lines.append(f" {row[0]:<15} {row[1]:<12} "
                        f"{'{row[2]}:>8} {'{row[3]}:>8} {'{row[4]}:>6}")
    else:
        for item in data:
            lines.append(f" {item}")
    return "\n".join(lines)

def build_report(args):
    lines = []
    lines.append("=" * 60)
    lines.append(" SERVER INVENTORY REPORT")
    lines.append(f" Generated: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}")
    lines.append("=" * 60)

    lines.append(format_section("SYSTEM INFORMATION", collect_system()))
    lines.append(format_section("CPU INFORMATION", collect_cpu()))

```

```

lines.append(format_section("MEMORY INFORMATION", collect_memory()))

lines.append(format_section("DISK INFORMATION", collect_disk()))


# Network

net = collect_network()

totals = net.pop("_totals", {})

lines.append(f"\n {'='*55}\n NETWORK INTERFACES\n {'-'*55}")

for iface, info in net.items():

    lines.append(f" {iface:<12}: {info['IP']:<18} "
                f"Mask: {info['Netmask']:<18} [{info['Status']}]")


for k, v in totals.items():

    lines.append(f" {k:<22}: {v}")


# Top processes

lines.append(f"\n {'='*55}\n TOP 10 PROCESSES BY MEMORY\n {'-'*55}")

lines.append(f" {'PID':<8} {'NAME':<22} {'USER':<12} {'MEM%':>6} {'CPU%':>6}")

lines.append(f" {'-'*55}")

for p in collect_processes():

    lines.append(f" {p['pid']:<8} {str(p['name']):<22} "
                f"{str(p['username']):<12} "
                f"{p['memory_percent']:>5.1f}% "
                f"{p['cpu_percent']:>5.1f}%")



lines.append(f"\n{'='*60}\n")

return "\n".join(lines)


def main():

    args = build_parser().parse_args()

    log.info("Collecting inventory...")

```

```
report = build_report(args)
print(report)

if args.output:
    Path(args.output).write_text(report)
    log.info(f"Report saved: {args.output}")
```

```
if __name__ == "__main__":
    main()
```

Scheduling Python Scripts with Cron

```
bash

# Install required libraries first
pip3 install psutil paramiko schedule --break-system-packages

# Edit crontab
crontab -e

# Run health monitor every 5 minutes
*/5 * * * * python3 /opt/scripts/health_monitor.py

# Run backup daily at 2 AM
0 2 * * * python3 /opt/scripts/auto_backup.py /var/www

# Analyze auth log every hour
0 * * * * python3 /opt/scripts/log_analyzer_pro.py \
    --output /tmp/security_report.txt

# Run user audit daily at 6 AM
0 6 * * * python3 /opt/scripts/user_audit.py \
```

```
--output /tmp/user_audit.txt

# Run inventory every Monday at 7 AM
0 7 * * 1 python3 /opt/scripts/server_inventory.py \
--output /tmp/inventory.txt
```

Python Course Complete!

Here's everything you've mastered:

Module	Topic	Key Skills
1	Foundations	Variables, types, f-strings, sys.argv
2	Control Flow	if/elif, for/while, list comprehensions
3	Functions	Args, returns, docstrings, logging module
4	Data Structures	Lists, dicts, tuples, sets
5	Files & OS	open(), pathlib, shutil, subprocess
6	Error Handling	try/except, custom exceptions, contextmanager
7	Sysadmin Libraries	argparse, psutil, paramiko, schedule
8	Real Projects	5 production scripts ready to deploy

Shall we start Ansible / RHCE? 

Can you generate a PDF file for python part printing all the above data as it is in the same format and colour with missing any thing with out any changes