🐍 Scripting & Automation

Days 6-7: DevOps Automation Through Code

Shell Scripting, Python Programming & Configuration Management

Training Period: 23rd July - 19th September, 2025

Focus: Automation Scripts, Python for DevOps, YAML & Configuration as Code

🎯 "Automate Everything" - The DevOps Mantra

⚡ **Speed**
Scripts execute faster than humans

🎯 **Accuracy**
No human errors or typos

🔄 **Consistency**
Same result every time

Prepared by: Rashi Rana
(Corporate Trainer)

🗺️ Days 6-7: Learning Roadmap

🚀 "From Manual Tasks to Automated Solutions"

🐧 Day 6: Shell Scripting

- Linux fundamentals & file system
- Essential commands & navigation
- Bash scripting basics
- Variables, loops & conditions
- File operations & text processing
- System administration scripts

🐍 Day 6-7: Python for DevOps

- Python syntax & data structures
- File handling & system operations
- API interactions & web requests
- Error handling & logging
- DevOps libraries & modules
- Automation script development

📄 Day 7: YAML & Configuration

- YAML syntax & structure
- Configuration file management
- CI/CD pipeline configurations
- Docker Compose files
- Kubernetes manifests
- Best practices & validation

🛠️ Hands-on Projects

- Server health monitoring script
- Automated backup solution
- Log analysis & reporting
- API integration project
- Configuration management
- DevOps workflow automation

🎓 By the End of Days 6-7, You Will:

✅ Write powerful shell scripts for system automation
✅ Develop Python programs for DevOps tasks
✅ Handle files, APIs, and system operations
✅ Create and manage YAML configurations
✅ Build automated monitoring solutions
✅ Implement configuration as code practices

🐧 Linux Fundamentals: The DevOps Foundation

🌍 "90% of cloud servers run Linux - Master the foundation!"

🤔 Why Linux Dominates DevOps?

📊 Market Reality:

- 96% of top 1 million web servers
- 100% of supercomputers worldwide
- 85% of smartphones (Android)
- All major cloud platforms

💪 DevOps Advantages:

- Free and open source
- Powerful command-line tools
- Excellent for automation
- Stable and secure

🗂️ Linux File System: Your Digital Filing Cabinet

🏠
/
Root directory
"The main cabinet"

👤
/home
User directories
"Your personal drawer"

⚙️
/etc
Configuration files
"The settings manual"

📊
/var
Variable data
"Logs & databases"

🔧
/usr
User programs
"Application storage"

💾
/tmp
Temporary files
"Scratch paper"

⌨️ Essential Linux Commands

🧭 Navigation

pwd – where am I?

ls – what's here?

cd – go somewhere

📁 File Operations

cp – copy files

mv – move/rename

rm – delete files

👀 File Viewing

cat – show file content

less – page through file

head/tail – first/last lines

Prepared by: Rashi Rana
(Corporate Trainer)

🔍 Essential Linux Text Processing Commands

📚 "Understanding the tools before using them"

🔍 grep - Global Regular Expression Print

🤔 What does grep do?

grep searches for specific patterns (text) in files or input. Think of it as "Find" function in a text editor, but much more powerful.

💡 Simple Analogy:
Like using Ctrl+F to find words in a document, but grep can search through thousands of files at once!

📝 Common grep Examples:

```
# Find lines containing "ERROR"
grep "ERROR" logfile.txt

# Case-insensitive search
grep -i "error" logfile.txt

# Count matching lines
grep -c "WARNING" logfile.txt

# Show line numbers
grep -n "INFO" logfile.txt

# Search in multiple files
grep "password" *.conf
```

📊 sort - Arrange Lines in Order

🤔 What does sort do?

sort arranges lines of text in alphabetical, numerical, or custom order. Essential for organizing data.

💡 Simple Analogy:
Like organizing a deck of cards - you can sort by number, suit, or any pattern you want!

📝 Common sort Examples:

```
# Sort lines alphabetically
sort names.txt

# Sort numbers properly
sort -n numbers.txt

# Reverse sort (Z to A)
sort -r names.txt

# Sort by specific column
sort -k2 data.txt

# Remove duplicates while sorting
sort -u list.txt
```

🎯 uniq - Remove or Count Duplicates

🤔 What does uniq do?

uniq removes duplicate lines or counts how many times each line appears. Works best with sorted data.

💡 Simple Analogy:
Like removing duplicate contacts from your phone book, or counting how many times each name appears!

📝 Common uniq Examples:

```
# Remove duplicate lines
uniq data.txt

# Count occurrences
uniq -c data.txt

# Show only duplicates
uniq -d data.txt

# Show only unique lines
uniq -u data.txt

# Common pattern: sort then uniq
sort data.txt | uniq -c
```

✂️ cut & sed - Text Extraction and Editing

✂️ cut - Extract Columns

Extracts specific columns or characters from each line.

```
# Extract 1st column (space-separated)
cut -d' ' -f1 file.txt

# Extract characters 1-5
cut -c1-5 file.txt

# Extract multiple columns
cut -d',' -f1,3 data.csv
```

🔧 sed - Stream Editor

Powerful tool for find and replace operations.

```
# Replace first occurrence
sed 's/old/new/' file.txt

# Replace all occurrences
sed 's/old/new/g' file.txt

# Delete lines containing pattern
sed '/pattern/d' file.txt
```

Prepared by: Rashi Rana
(Corporate Trainer)

⌨️ Hands-On Linux Commands Workshop

🎯 **"Practice makes perfect - Let's get our hands dirty!"**

🧭 Exercise 1: Navigation & File Exploration

📝 Commands to Try:

```
# Find out where you are
pwd

# See what's in current directory
ls

# See detailed file information
ls -la

# See hidden files
ls -a

# Navigate to home directory
cd ~

# Go to root directory
cd /

# Go back to previous directory
cd -

# Navigate to parent directory
cd ..
```

🎯 What You'll Learn:

- **pwd** - Shows current directory path
- **ls** - Lists files and folders
- **ls -la** - Shows permissions, size, date
- **cd ~** - Goes to your home folder
- **cd /** - Goes to system root
- **cd -** - Goes to previous location
- **cd ..** - Goes up one level

💡 **Pro Tip:** Use Tab key for auto-completion!

📁 Exercise 2: File & Directory Operations

🚀 Step-by-Step Practice:

```
# Create a practice directory
mkdir devops_practice

cd devops_practice

# Create some files
touch server1.log server2.log config.txt

# Create a file with content
echo "Hello DevOps World!" > welcome.txt

echo "This is line 2" >> welcome.txt

# Create a subdirectory
mkdir logs backup
```

```
# Copy files
cp welcome.txt backup/

cp *.log logs/

# List everything
ls -la

ls logs/

ls backup/

# Move and rename files
mv config.txt settings.conf

mv settings.conf backup/

# View file contents
cat welcome.txt

head -n 1 welcome.txt

tail -n 1 welcome.txt
```

📄 Exercise 3: Text Processing & Search

📝 Create Sample Log File:

```
# Create a sample log file
cat > sample.log << EOF
2024-01-15 10:30:15 INFO Server started
2024-01-15 10:30:20 INFO Database
connected
2024-01-15 10:35:10 ERROR Connection
timeout
2024-01-15 10:35:15 WARN Retrying
connection
2024-01-15 10:35:20 INFO Connection
restored
2024-01-15 10:40:05 ERROR Disk space low
2024-01-15 10:45:30 INFO Backup completed
EOF
```

🔍 Practice Text Commands:

```
# Search for errors
grep "ERROR" sample.log

# Count lines
wc -l sample.log

# Count words
wc -w sample.log

# Show first 3 lines
head -n 3 sample.log

# Show last 2 lines
tail -n 2 sample.log

# Search and count
grep -c "INFO" sample.log

# Case insensitive search
grep -i "error" sample.log
```

🔐 Linux Permissions & System Commands

🛡️ "Understanding who can do what with files"

🔍 Understanding File Permissions

📊 Permission Structure:

```
# Example: -rwxr-xr--
# Position: -rwx r-x r--
#           |  |   |   |
#           |  |   |   └── Others
(everyone else)
#           |  |   └────── Group (file's
group)
#           |  └────────── Owner (file
creator)
#           └───────────── File type (- =
file, d = directory)

# Permission meanings:
# r = read (4)
# w = write (2)
# x = execute (1)
```

🎯 Common Permission Patterns:

```
755 - rwxr-xr-x (executable files)
644 - rw-r--r-- (regular files)
600 - rw------- (private files)
777 - rwxrwxrwx (full access -
dangerous!)
```

🛠️ Exercise 4: Working with Permissions

📝 Create and Check Permissions:

```
# Create test files
touch public_file.txt

touch private_file.txt

touch script.sh

# Check current permissions
ls -la

# View permissions in detail
ls -l public_file.txt

# Check who you are
whoami

groups
```

🔧 Modify Permissions:

```
# Make file private (owner only)
chmod 600 private_file.txt

# Make script executable
chmod +x script.sh

# or alternatively
chmod 755 script.sh

# Make file readable by everyone
chmod 644 public_file.txt

# Remove write permission for group/others
chmod go-w public_file.txt

# Check changes
ls -la
```

💻 Exercise 5: System Information Commands

📊 System Status:

```
# Check system uptime
uptime

# See current date and time
date

# Check disk usage
df -h

# Check memory usage
free -h

# See running processes
ps aux | head -10

# Check current users
who

# System information
uname -a
```

🔍 Process Management:

```
# Find processes by name
ps aux | grep bash

# Check CPU usage
top -n 1

# Find files by name
find . -name "*.txt"

# Find files by size
find . -size +1M

# Check network connections
netstat -tuln

# Check environment variables
env | head -5
```

🎯 Challenge Exercise:

Create a directory structure for a web application and set appropriate permissions:

```
# Create directory structure
mkdir -p webapp/{public,private,logs,config}

# Create sample files
touch webapp/public/index.html

touch webapp/private/database.conf

touch webapp/logs/app.log

touch webapp/config/settings.ini

# Set appropriate permissions
chmod 755 webapp/public/

chmod 644 webapp/public/index.html

chmod 700 webapp/private/

chmod 600 webapp/private/database.conf

chmod 755 webapp/logs/

chmod 644 webapp/logs/app.log

# Verify permissions
ls -la webapp/

ls -la webapp/*/
```

🔧 Linux Text Processing & Pipes Mastery

🚰 **"Connecting commands like water through pipes"**

🚰 Understanding Pipes & Redirection

📝 Pipe Concept:

Pipes (|) take output from one command and send it as input to another command.

```
# Basic pipe example
command1 | command2 | command3

# Real example
ls -la | grep ".txt" | wc -l

# This means:
# 1. ls -la (list files)
# 2. | grep ".txt" (filter for .txt
files)
# 3. | wc -l (count lines)
```

📤 Redirection Operators:

```
> - Redirect output (overwrite)
>> - Redirect output (append)
< - Redirect input
2> - Redirect errors
&> - Redirect all output
```

📊 Exercise 6: Log Analysis with Pipes

🗂️ First, Create a Realistic Log File:

```
# Create a comprehensive log file
cat > server.log << 'EOF'
2024-01-15 08:30:15 INFO [web-01] Server started successfully
2024-01-15 08:30:20 INFO [db-01] Database connection established
2024-01-15 08:35:10 ERROR [web-01] Connection timeout to database
2024-01-15 08:35:15 WARN [web-01] Retrying database connection
2024-01-15 08:35:20 INFO [web-01] Database connection restored
2024-01-15 08:40:05 ERROR [disk] Disk space low on /var partition
2024-01-15 08:45:30 INFO [backup] Daily backup completed successfully
2024-01-15 09:15:45 ERROR [web-02] Memory usage critical: 95%
2024-01-15 09:20:10 WARN [web-02] High CPU usage detected: 85%
2024-01-15 09:25:30 INFO [web-02] System performance normalized
2024-01-15 10:30:15 ERROR [api] Rate limit exceeded for user 12345
2024-01-15 10:35:20 INFO [api] User authentication successful
EOF
```

🔍 Now Practice Text Processing:

```
# Count total log entries
wc -l server.log
```

```
# Get unique server names
grep -o '\[.*\]' server.log | sort | uniq
```

```
# Find all ERROR entries
grep "ERROR" server.log

# Count ERROR entries
grep -c "ERROR" server.log

# Find errors and warnings
grep -E "(ERROR|WARN)" server.log

# Show only timestamps and messages
cut -d' ' -f1,2,4- server.log
```

```
# Find web server issues
grep "web-" server.log

# Show last 5 log entries
tail -n 5 server.log

# Show first 3 entries
head -n 3 server.log

# Search case-insensitive
grep -i "database" server.log
```

🚀 Exercise 7: Advanced Pipe Combinations

🎯 Real DevOps Scenarios:

📈 Log Analysis Pipelines:

```
# Find top error sources
grep "ERROR" server.log | \
  cut -d' ' -f4 | \
  sort | uniq -c | \
  sort -nr

# Get hourly error count
grep "ERROR" server.log | \
  cut -d' ' -f2 | \
  cut -d':' -f1 | \
  sort | uniq -c

# Find critical memory issues
grep "Memory usage critical" server.log | \
  wc -l
```

🔍 System Monitoring:

```
# Find largest files
ls -la | sort -k5 -nr | head -5

# Count files by extension
ls -1 | grep '\.' | \
  sed 's/.*\.//' | \
  sort | uniq -c | \
  sort -nr

# Monitor disk usage
df -h | grep -v "tmpfs" | \
  awk '{print $5 " " $6}' | \
  sort -nr
```

🏆 Master Challenge:

Create a one-liner to find the most active hour in the log file:

```
# Solution: Extract hour, count occurrences, sort by count
cat server.log | \
  cut -d' ' -f2 | \
  cut -d':' -f1 | \
  sort | uniq -c | \
  sort -nr | \
  head -1
```

Explanation: This pipeline extracts the hour from each log entry, counts how many times each hour appears, sorts by count (highest first), and shows the top result.

Prepared by: Rashi Rana
(Corporate Trainer)

📜 Shell Scripting: Automating Linux Tasks

🤖 **"Turn repetitive commands into powerful scripts"**

🤔 What is Shell Scripting?

Simple Definition: A shell script is a file containing a series of commands that the shell can execute automatically.

❌ Without Scripts:

- Type same commands repeatedly
- Risk of typos and errors
- Time-consuming manual work
- Hard to share procedures

✅ With Scripts:

- Run complex tasks with one command
- Consistent execution every time
- Save time and reduce errors
- Easy to share and version control

🚀 Your First Shell Script

📝 hello_devops.sh

```
#!/bin/bash
# This is a comment - my first DevOps script!

echo "Hello, DevOps World!"
echo "Today is: $(date)"
echo "Current user: $(whoami)"
echo "Current directory: $(pwd)"

# Let's check system info
echo "System uptime:"
uptime
```

🏃 How to run it:

```
chmod +x hello_devops.sh
./hello_devops.sh
```

🧩 Shell Script Components

📋 Essential Elements:

🔧 Advanced Features:

- **#!/bin/bash** - Shebang line
- **# comments** - Documentation
- **echo** - Print output
- **$(command)** - Command substitution

- **Variables** - Store data
- **Loops** - Repeat actions
- **Conditions** - Make decisions
- **Functions** - Reusable code

Prepared by: Rashi Rana
(Corporate Trainer)

🔧 Advanced Shell Scripting: Variables, Loops & Logic

🧠 "Making scripts smart and dynamic"

📦 Variables: Storing Information

📝 Basic Variables:

```
#!/bin/bash

# Define variables
SERVER_NAME="web-server-01"
PORT=8080
LOG_FILE="/var/log/app.log"

# Use variables
echo "Checking $SERVER_NAME"
echo "Port: $PORT"
echo "Log file: $LOG_FILE"
```

🔄 Command Variables:

```
#!/bin/bash

# Store command output
CURRENT_DATE=$(date +%Y-%m-%d)
DISK_USAGE=$(df -h / | tail -1)
USER_COUNT=$(who | wc -l)

echo "Date: $CURRENT_DATE"
echo "Disk: $DISK_USAGE"
echo "Users online: $USER_COUNT"
```

🔄 Loops & Conditions: Smart Automation

🔁 For Loop Example:

```
#!/bin/bash

# Check multiple servers
SERVERS=("web1" "web2" "web3")

for server in "${SERVERS[@]}"; do
    echo "Pinging $server..."
    if ping -c 1 $server &> /dev/null;
then
        echo "✅ $server is UP"
    else
        echo "❌ $server is DOWN"
    fi
done
```

❓ If-Else Logic:

```
#!/bin/bash

# Check disk space
DISK_USAGE=$(df / | tail -1 | awk '{print
$5}' | sed 's/%//')

if [ $DISK_USAGE -gt 80 ]; then
    echo "🚨 ALERT: Disk usage is
${DISK_USAGE}%"
    echo "Cleaning up logs..."
    # Add cleanup commands here
elif [ $DISK_USAGE -gt 60 ]; then
    echo "⚠️ WARNING: Disk usage is
${DISK_USAGE}%"
else
    echo "✅ Disk usage is healthy:
${DISK_USAGE}%"
fi
```

🚀 Real DevOps Script: Server Health Check

```bash
#!/bin/bash
# server_health_check.sh - Complete server monitoring script

LOG_FILE="/var/log/health_check.log"
DATE=$(date '+%Y-%m-%d %H:%M:%S')

echo "[$DATE] Starting server health check..." | tee -a $LOG_FILE

# Check CPU usage
CPU_USAGE=$(top -bn1 | grep "Cpu(s)" | awk '{print $2}' | sed 's/%us,//')
echo "CPU Usage: $CPU_USAGE%" | tee -a $LOG_FILE

# Check memory usage
MEMORY_USAGE=$(free | grep Mem | awk '{printf("%.2f%%", $3/$2 * 100.0)}')
echo "Memory Usage: $MEMORY_USAGE" | tee -a $LOG_FILE

# Check disk space
DISK_USAGE=$(df -h / | tail -1 | awk '{print $5}')
echo "Disk Usage: $DISK_USAGE" | tee -a $LOG_FILE

# Check running services
SERVICES=("nginx" "mysql" "redis")
for service in "${SERVICES[@]}"; do
    if systemctl is-active --quiet $service; then
        echo "✅ $service is running" | tee -a $LOG_FILE
    else
        echo "❌ $service is not running" | tee -a $LOG_FILE
    fi
done

echo "[$DATE] Health check completed!" | tee -a $LOG_FILE
```

Prepared by: Rashi Rana
(Corporate Trainer)

🐍 Python for DevOps: The Automation Powerhouse

🚀 "Simple syntax, powerful capabilities"

🤔 Why Python Dominates DevOps?

📊 Industry Facts:

- Most popular language for automation
- Used by Netflix, Google, Instagram
- Huge ecosystem of DevOps libraries
- Easy to learn and maintain

💪 DevOps Superpowers:

- API integrations made simple
- Powerful text processing
- Cross-platform compatibility
- Rich standard library

⚖️ Python vs Shell: When to Use What?

🐚 Use Shell When:

- Simple file operations
- Chaining Linux commands
- Quick one-liners
- System administration tasks
- Startup scripts

🐍 Use Python When:

- Complex data processing
- API interactions
- Error handling needed
- Cross-platform scripts
- Advanced automation

🎯 Python Basics You Need for DevOps

📦 Data Types

```
strings = "text"
numbers = 42
lists = [1, 2, 3]
dicts = {"key": "value"}
```

🔄 Control Flow

```
if condition:
  do_something()
for item in list:
  process(item)
```

📚 Libraries

```
import os
import requests
import json
import subprocess
```

Prepared by: Rashi Rana
(Corporate Trainer)

🐍 Python Variables & Fundamentals

📊 Building Blocks of Python Programming

🏷️ Variable Declaration

```
# Python variables are dynamically typed
name = "DevOps Engineer"
age = 25
salary = 75000.50
is_certified = True

# Multiple assignment
x, y, z = 1, 2, 3
a = b = c = 100
```

📋 Data Types

```
# Basic data types
text = "Hello World"        # str
number = 42                 # int
decimal = 3.14              # float
flag = True                 # bool
items = [1, 2, 3]           # list
config = {"env": "prod"}    # dict
coords = (10, 20)           # tuple
```

💬 Input/Output

```
# Getting user input
name = input("Enter your name: ")
age = int(input("Enter age: "))

# Output with formatting
print(f"Hello {name}!")
print(f"You are {age} years old")

# Multiple outputs
print("Server:", "nginx", "Status:", "running")
```

🔀 Conditional Statements

```
# Simple conditions
server_status = "running"

if server_status == "running":
    print("✅ Server is healthy")
elif server_status == "stopped":
    print("⚠️ Server is down")
else:
    print("❓ Unknown status")
```

🔄 Loops

```
# For loop
servers = ["web1", "web2", "db1"]
for server in servers:
    print(f"Checking {server}...")

# While loop
count = 0
while count < 3:
    print(f"Attempt {count + 1}")
    count += 1

# Range loop
for i in range(1, 6):
    print(f"Port 808{i}")
```

⚙️ Functions & Reusability

```
# Function definition
def check_server_health(server_name, port=80):
    """Check if server is responding"""
    print(f"Checking {server_name}:{port}")
    return "healthy"

# Function usage
status = check_server_health("web-server")
result = check_server_health("api-server", 8080)

# Lambda functions
square = lambda x: x ** 2
print(square(5))  # Output: 25
```

💡 **Key Takeaway**

Python's simplicity and readability make it perfect for DevOps automation tasks!

🔄 Python Program Execution Flow

📝 "Understanding how Python executes your code step by step"

📝 Simple Python Program Example

💻 **server_health.py**

```
#!/usr/bin/env python3
"""
Simple Server Health Check Program
Demonstrates basic Python concepts and execution flow
"""

# Step 1: Import required modules
import datetime
import random

# Step 2: Define variables
server_name = "web-server-01"
max_cpu_threshold = 80
max_memory_threshold = 75

# Step 3: Define functions
def get_current_time():
    """Get current timestamp"""
    return datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")

def simulate_server_metrics():
    """Simulate server CPU and memory usage"""
    cpu_usage = random.randint(20, 95)
    memory_usage = random.randint(30, 90)
    return cpu_usage, memory_usage

def check_server_health(cpu, memory, cpu_limit, memory_limit):
    """Check if server metrics are within acceptable limits"""
    status = "HEALTHY"
    alerts = []

    if cpu > cpu_limit:
        status = "WARNING"
        alerts.append(f"High CPU usage: {cpu}%")

    if memory > memory_limit:
        status = "CRITICAL" if status == "WARNING" else "WARNING"
        alerts.append(f"High memory usage: {memory}%")

    return status, alerts

def display_results(server, timestamp, cpu, memory, status, alerts):
    """Display the health check results"""
    print("=" * 50)
    print("🖥️  SERVER HEALTH REPORT")
    print("=" * 50)
    print(f"Server: {server}")
    print(f"Timestamp: {timestamp}")
    print(f"CPU Usage: {cpu}%")
    print(f"Memory Usage: {memory}%")
    print(f"Status: {status}")

    if alerts:
        print("\n🚨  ALERTS:")
        for alert in alerts:
            print(f"  - {alert}")
    else:
        print("\n✅ All systems normal")

# Step 4: Main program execution
def main():
    """Main function - program entry point"""
    print("🚀 Starting server health check...")

    # Get current time
    current_time = get_current_time()

    # Get server metrics
    cpu_usage, memory_usage = simulate_server_metrics()

    # Check health status
    health_status, alert_list = check_server_health(
        cpu_usage,
        memory_usage,
        max_cpu_threshold,
        max_memory_threshold
    )

    # Display results
    display_results(
        server_name,
        current_time,
        cpu_usage,
        memory_usage,
        health_status,
        alert_list
    )

    print("\n🏁 Health check completed!")

# Step 5: Program entry point
if __name__ == "__main__":
    main()
```

🔄 Step-by-Step Execution Flow

📋 Execution Order:

1. Python reads the entire file
2. Imports modules (datetime, random)
3. Creates variables (server_name, thresholds)
4. Defines functions (stores in memory)
5. Checks if __name__ == "__main__"
6. Calls main() function
7. Executes main() step by step
8. Program ends

🎯 What Happens in main():

A. Print "Starting health check..."
B. Call get_current_time()
C. Call simulate_server_metrics()
D. Call check_server_health()
E. Call display_results()
F. Print "Health check completed!"
G. Return to Python (program ends)

📊 Visual Execution Flow

START
Python loads
the script

SETUP
Import modules
Define variables

EXECUTE
Run main()
Call functions

END
Display results
Program exits

🎯 Key Execution Concepts:

- **Sequential execution:** Python runs code line by line, top to bottom
- **Function definitions:** Functions are stored in memory but not executed until called
- **Main guard:** `if __name__ == "__main__":` ensures main() only runs when script is executed directly
- **Function calls:** When a function is called, Python jumps to that function, executes it, then returns
- **Variable scope:** Variables defined in functions are local, global variables are accessible everywhere

Prepared by: Rashi Rana
(Corporate Trainer)

☁️ Python for AWS Lambda & Cloud Automation

🚀 "Serverless automation with Python - the DevOps game changer"

🤔 What is AWS Lambda?

📝 Simple Definition:

AWS Lambda lets you run Python code in the cloud without managing servers. You just upload your code, and AWS runs it automatically when triggered.

💡 Think of it as:
A Python function that lives in the cloud and wakes up only when needed!

🎯 Key Benefits:

- **No servers to manage** - AWS handles everything
- **Pay per use** - Only charged when code runs
- **Auto-scaling** - Handles any amount of traffic
- **Event-driven** - Responds to triggers automatically
- **Fast deployment** - Upload code and it's live

🔧 Real AWS Lambda Example: Automated Backup

📝 lambda_backup.py - Automatic EC2 Snapshot Creation

```python
import json
import boto3
from datetime import datetime, timedelta

def lambda_handler(event, context):
    """
    AWS Lambda function to automatically create EC2 snapshots
    Triggered daily by CloudWatch Events
    """

    # Initialize AWS clients
    ec2 = boto3.client('ec2')

    try:
        # Get all EC2 instances with backup tag
        response = ec2.describe_instances(
            Filters=[
                {
                    'Name': 'tag:AutoBackup',
                    'Values': ['true']
                },
                {
                    'Name': 'instance-state-name',
                    'Values': ['running', 'stopped']
                }
            ]
        )

        snapshots_created = 0

        # Process each instance
        for reservation in response['Reservations']:
            for instance in reservation['Instances']:
                instance_id = instance['InstanceId']
                instance_name = 'Unknown'

                # Get instance name from tags
                for tag in instance.get('Tags', []):
                    if tag['Key'] == 'Name':
                        instance_name = tag['Value']
                        break

                # Create snapshots for each volume
                for volume in instance.get('BlockDeviceMappings', []):
                    volume_id = volume['Ebs']['VolumeId']

                    # Create snapshot
                    snapshot_description = f"Auto backup of {instance_name} ({instance_id}) -
{datetime.now().strftime('%Y-%m-%d %H:%M:%S')}"

                    snapshot = ec2.create_snapshot(
                        VolumeId=volume_id,
                        Description=snapshot_description
                    )

                    # Tag the snapshot
                    ec2.create_tags(
                        Resources=[snapshot['SnapshotId']],
                        Tags=[
                            {'Key': 'Name', 'Value': f"Auto-backup-{instance_name}"},
                            {'Key': 'InstanceId', 'Value': instance_id},
                            {'Key': 'CreatedBy', 'Value': 'Lambda-AutoBackup'},
                            {'Key': 'DeleteAfter', 'Value': (datetime.now() +
timedelta(days=7)).strftime('%Y-%m-%d')}
                        ]
                    )

                    snapshots_created += 1
                    print(f"Created snapshot {snapshot['SnapshotId']} for instance
{instance_name}")

        # Clean up old snapshots (older than 7 days)
        cleanup_old_snapshots(ec2)

        return {
            'statusCode': 200,
            'body': json.dumps({
                'message': f'Successfully created {snapshots_created} snapshots',
                'snapshots_created': snapshots_created,
                'timestamp': datetime.now().isoformat()
            })
        }

    except Exception as e:
        print(f"Error: {str(e)}")
        return {
            'statusCode': 500,
            'body': json.dumps({
                'error': str(e),
                'timestamp': datetime.now().isoformat()
            })
        }

def cleanup_old_snapshots(ec2_client):
    """Remove snapshots older than retention period"""
    try:
        # Get snapshots created by this Lambda
        snapshots = ec2_client.describe_snapshots(
            OwnerIds=['self'],
            Filters=[
                {'Name': 'tag:CreatedBy', 'Values': ['Lambda-AutoBackup']}
            ]
        )

        deleted_count = 0
        today = datetime.now().date()

        for snapshot in snapshots['Snapshots']:
            # Check DeleteAfter tag
            delete_after = None
            for tag in snapshot.get('Tags', []):
                if tag['Key'] == 'DeleteAfter':
                    delete_after = datetime.strptime(tag['Value'], '%Y-%m-%d').date()
                    break

            # Delete if past retention date
            if delete_after and today > delete_after:
                ec2_client.delete_snapshot(SnapshotId=snapshot['SnapshotId'])
                print(f"Deleted old snapshot: {snapshot['SnapshotId']}")
                deleted_count += 1

        print(f"Cleaned up {deleted_count} old snapshots")

    except Exception as e:
        print(f"Cleanup error: {str(e)}")
```

🤖 Python Automation Use Cases in DevOps

☁️ Cloud Automation:

- **Auto-scaling:** Scale resources based on demand
- **Cost optimization:** Stop unused instances
- **Security compliance:** Audit and fix security groups
- **Backup automation:** Scheduled snapshots and backups
- **Log processing:** Analyze CloudWatch logs
- **Resource tagging:** Automatically tag resources

🔧 Infrastructure Automation:

- **CI/CD pipelines:** Automated deployments
- **Configuration management:** Update server configs
- **Monitoring alerts:** Custom alerting systems
- **Database maintenance:** Automated DB tasks
- **File processing:** Batch file operations
- **API integrations:** Connect different services

⚡ How Lambda Functions Get Triggered

⏰

Scheduled Events

CloudWatch Events trigger functions on schedule (daily backups, weekly reports)

📁

File Uploads

S3 bucket events trigger processing when files are uploaded

🌐

API Calls

API Gateway triggers functions when HTTP requests are made

💾

Database Changes

DynamoDB streams trigger functions when data changes

📧

Messages

SQS/SNS messages trigger functions for event processing

🚨

Alerts

CloudWatch alarms trigger functions for automated responses

🎯 Real-World Lambda Automation Examples:

- **Image processing:** Resize uploaded images automatically
- **Log analysis:** Process CloudWatch logs for security events
- **Data transformation:** Convert CSV to JSON when files are uploaded
- **Notification system:** Send Slack alerts for system events
- **Auto-remediation:** Fix security group violations automatically
- **Cost monitoring:** Alert when AWS spending exceeds budget
- **Compliance checking:** Ensure resources follow company policies
- **Data backup:** Automated database and file backups

🎯 Python Practice Exercises

💪 **"Practice makes perfect - Let's code!"**

🚀 Exercise 1: Website Status Checker

📝 Task: Check website accessibility and response times

```
#!/usr/bin/env python3
"""
Exercise 1: Website Status Checker
Cross-platform website monitoring tool
"""

import requests
import time
import platform
from datetime import datetime

# List of websites to monitor
websites = [
    {"name": "Google", "url": "https://www.google.com"},
    {"name": "GitHub", "url": "https://github.com"},
    {"name": "Stack Overflow", "url": "https://stackoverflow.com"},
    {"name": "Python.org", "url": "https://www.python.org"},
    {"name": "AWS", "url": "https://aws.amazon.com"}
]

def check_website(site_info, timeout=10):
    """Check website accessibility and measure response time"""
    try:
        print(f"🔍 Checking {site_info['name']}...")

        start_time = time.time()
        response = requests.get(site_info['url'], timeout=timeout)
        response_time = round((time.time() - start_time) * 1000, 2)

        if response.status_code == 200:
            print(f"✅ {site_info['name']} - OK ({response_time}ms)")
            return {
                'status': 'UP',
                'response_time': response_time,
                'status_code': response.status_code
            }
        else:
            print(f"⚠️ {site_info['name']} - Status: {response.status_code}")
            return {
                'status': 'WARNING',
                'response_time': response_time,
                'status_code': response.status_code
            }

    except requests.exceptions.Timeout:
        print(f"⏰ {site_info['name']} - Request timeout")
        return {'status': 'TIMEOUT', 'response_time': timeout * 1000}

    except requests.exceptions.ConnectionError:
        print(f"🔌 {site_info['name']} - Connection failed")
        return {'status': 'DOWN', 'response_time': 0}

    except Exception as e:
        print(f"❌ {site_info['name']} - Error: {str(e)}")
        return {'status': 'ERROR', 'response_time': 0}
```

```
def main():
    """Main monitoring function"""
    print("🌐 WEBSITE STATUS CHECKER")
    print("=" * 40)
    print(f"🖥️  Platform: {platform.system()}")
    print(f"⏰ Check time: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}")
    print()

    results = []
    for site in websites:
        result = check_website(site)
        results.append({'name': site['name'], 'result': result})
        print()
        time.sleep(0.5)

    # Summary
    up_sites = len([r for r in results if r['result']['status'] == 'UP'])
    print(f"📊 Summary: {up_sites}/{len(results)} sites are UP")

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

🎯 Setup Instructions:

Windows:

1. Install: `pip install requests`
2. Save as `website_checker.py`
3. Run: `python website_checker.py`

Mac:

1. Install: `pip3 install requests`
2. Save as `website_checker.py`
3. Run: `python3 website_checker.py`

📁 Exercise 2: System Information Script

📝 Task: Create a cross-platform system monitor

```
#!/usr/bin/env python3
"""
Exercise 2: System Information Script
Cross-platform system monitoring tool
"""

import os
import platform
import shutil
from datetime import datetime
from pathlib import Path

def get_system_info():
    """Get basic system information"""
    system = platform.system()
    if system == "Darwin":
        os_name = "macOS"
    elif system == "Windows":
        os_name = "Windows"
    elif system == "Linux":
        os_name = "Linux"
    else:
        os_name = system

    return {
        'os': os_name,
        'version': platform.release(),
        'architecture': platform.architecture()[0],
        'hostname': platform.node(),
        'python_version': platform.python_version()
    }
```

```
def get_disk_usage():
    """Get disk usage information"""
    try:
        if platform.system() == "Windows":
            # Windows - check C: drive
            total, used, free = shutil.disk_usage("C:\\")
        else:
            # Unix-like systems - check root
            total, used, free = shutil.disk_usage("/")

        # Convert to GB
        total_gb = round(total / (1024**3), 2)
        used_gb = round(used / (1024**3), 2)
        free_gb = round(free / (1024**3), 2)
        usage_percent = round((used / total) * 100, 1)

        return {
            'total': total_gb,
            'used': used_gb,
            'free': free_gb,
            'percentage': usage_percent
        }
    except Exception as e:
        return {'error': str(e)}

def main():
    """Main function"""
    print("🖥️  SYSTEM INFORMATION DASHBOARD")
    print("=" * 50)

    # System info
    sys_info = get_system_info()
    print(f"🏷️  Hostname: {sys_info['hostname']}")
    print(f"🖥️  Operating System: {sys_info['os']} {sys_info['version']}")
    print(f"🏗️  Architecture: {sys_info['architecture']}")
    print(f"🐍 Python Version: {sys_info['python_version']}")
    print(f"⏰ Current Time: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}")

    # Disk usage
    print(f"\n💿 DISK USAGE:")
    disk_info = get_disk_usage()
    if 'error' not in disk_info:
        print(f"📊 Total Space: {disk_info['total']} GB")
        print(f"💿 Used Space: {disk_info['used']} GB ({disk_info['percentage']}%)")
        print(f"📀 Free Space: {disk_info['free']} GB")

        if disk_info['percentage'] > 80:
            print("🚨 WARNING: Disk usage is high!")
    else:
        print(f"❌ Error getting disk info: {disk_info['error']}")

    print(f"\n✅ System check completed!")

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

🎯 Try This:

- Run the script on different operating systems
- Add more system information (CPU, memory)
- Save the report to a file
- Add error handling for different scenarios

📄 YAML Essentials: The Configuration Language

📝 "YAML Ain't Markup Language - Human-readable data serialization"

🤔 What is YAML?

📝 Simple Definition:

YAML is a human-readable data format used for configuration files, data exchange, and storing structured information.

💡 Think of it as:
A way to write configuration that both humans and computers can easily understand!

🎯 Why Use YAML?

- **Human-readable:** Easy to read and write
- **Widely supported:** Used by many DevOps tools
- **No brackets:** Uses indentation instead of { }
- **Comments allowed:** Document your configuration
- **Data types:** Strings, numbers, booleans, lists, objects

📚 YAML Syntax Basics

🔤 Basic Data Types:

```yaml
# Strings (quotes optional for simple
strings)
name: "John Doe"
city: New York
message: Hello World

# Numbers
age: 30
price: 99.99
count: 1000

# Booleans
is_active: true
is_admin: false
debug_mode: yes

# Null values
middle_name: null
nickname: ~
```

📋 Lists and Objects:

```yaml
# Lists (Arrays)
fruits:
  - apple
  - banana
  - orange

# Inline list
colors: [red, green, blue]

# Objects (Dictionaries)
person:
  name: Alice
  age: 25
  address:
    street: 123 Main St
    city: Boston
    zip: 02101

# Inline object
coordinates: {x: 10, y: 20}
```

⚠️ YAML Rules & Best Practices

📏 Critical Rules:

- **Indentation matters:** Use spaces, not tabs
- **Consistent spacing:** Same level = same indentation
- **Case sensitive:** 'Name' ≠ 'name'
- **Colons need space:** 'key: value' not 'key:value'
- **Comments start with #**

✅ Best Practices:

- **Use 2 spaces** for indentation
- **Quote strings** with special characters
- **Add comments** to explain complex sections
- **Keep it simple** - avoid deep nesting
- **Validate syntax** before using

❌ Common YAML Mistakes to Avoid

🚫 Wrong:

```
# Using tabs instead of spaces
name:    John Doe

# Inconsistent indentation
person:
  name: Alice
    age: 25

# Missing space after colon
database:localhost

# Unquoted special characters
message: Hello: World!
```

✅ Correct:

```
# Using spaces for indentation
name: John Doe

# Consistent indentation
person:
  name: Alice
  age: 25

# Space after colon
database: localhost

# Quoted special characters
message: "Hello: World!"
```

Prepared by: Rashi Rana
(Corporate Trainer)

⚙️ YAML Configuration Examples

🔧 **"Real-world YAML configurations you'll encounter"**

📱 Application Configuration File

📝 config/app.yml - Web Application Settings

```yaml
# Application Configuration File
app:
  name: "My Web Application"
  version: "1.2.0"
  environment: production
  debug: false

# Server Configuration
server:
  host: "0.0.0.0"
  port: 8080
  timeout: 30
  ssl:
    enabled: true
    certificate: "/etc/ssl/app.crt"
    private_key: "/etc/ssl/app.key"

# Database Settings
database:
  type: postgresql
  host: localhost
  port: 5432
  name: myapp_db
  username: app_user
  password: "${DB_PASSWORD}"
  pool_size: 10
  timeout: 5000

# Logging Configuration
logging:
  level: INFO
  format: json
  outputs:
    - type: file
      path: "/var/log/app.log"
      max_size: "100MB"
      max_files: 5
    - type: console
      enabled: true

# Feature Flags
features:
  user_registration: true
  email_verification: true
  two_factor_auth: false
  analytics: true
  maintenance_mode: false

# External Services
services:
  email:
    provider: sendgrid
    api_key: "${SENDGRID_API_KEY}"
    from_address: "noreply@myapp.com"

  cache:
    type: redis
    host: localhost
    port: 6379
    ttl: 3600

# Environment Variables
environment_variables:
  - NODE_ENV
  - DATABASE_URL
  - API_SECRET_KEY
  - REDIS_URL
```

🚀 CI/CD Pipeline Configuration

📝 .github/workflows/ci.yml - GitHub Actions

```
# GitHub Actions CI/CD Pipeline
name: Build and Test

# When to run this workflow
on:
  push:
    branches: [main, develop]
  pull_request:
    branches: [main]

# Environment variables for all jobs
env:
  NODE_VERSION: "18"
  PYTHON_VERSION: "3.9"

# Jobs to run
jobs:
  # Testing job
  test:
    name: Run Tests
    runs-on: ubuntu-latest

    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Setup Node.js
        uses: actions/setup-node@v3
        with:
          node-version: ${{ env.NODE_VERSION }}
          cache: npm

      - name: Install dependencies
        run: |
          npm ci
          npm audit fix

      - name: Run linting
        run: npm run lint

      - name: Run tests
        run: npm test -- --coverage

      - name: Upload test results
        uses: actions/upload-artifact@v3
        if: always()
        with:
          name: test-results
          path: test-results/

  # Build job
  build:
    name: Build Application
    runs-on: ubuntu-latest
    needs: test

    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Build application
        run: |
          npm ci
          npm run build

      - name: Save build artifacts
        uses: actions/upload-artifact@v3
        with:
          name: build-files
          path: dist/
          retention-days: 7
```

📊 Data & Inventory Configuration

👥 Team Directory:

```
# team.yml - Team Information
teams:
  - name: Development
    lead: John Smith
    members:
      - name: Alice Johnson
        role: Senior Developer
        skills: [Python, React, AWS]
      - name: Bob Wilson
        role: Junior Developer
        skills: [JavaScript, HTML, CSS]

  - name: DevOps
    lead: Sarah Davis
    members:
      - name: Mike Brown
        role: DevOps Engineer
        skills: [Docker, Kubernetes,
Terraform]
      - name: Lisa Chen
        role: Site Reliability Engineer
        skills: [Monitoring, Linux,
Python]

contact:
  email: team@company.com
  slack: "#dev-team"
  meeting_schedule:
    daily_standup: "09:00 AM"
    sprint_planning: "Monday 2:00 PM"
    retrospective: "Friday 4:00 PM"
```

🖥️ Server Inventory:

```
# inventory.yml - Server Configuration
servers:
  web_servers:
    - hostname: web-01
      ip: 192.168.1.10
      os: Ubuntu 20.04
      cpu: 4
      memory: 8GB
      disk: 100GB
      services: [nginx, nodejs]

    - hostname: web-02
      ip: 192.168.1.11
      os: Ubuntu 20.04
      cpu: 4
      memory: 8GB
      disk: 100GB
      services: [nginx, nodejs]

  database_servers:
    - hostname: db-01
      ip: 192.168.1.20
      os: Ubuntu 20.04
      cpu: 8
      memory: 16GB
      disk: 500GB
      services: [postgresql]
      backup_schedule: "daily at 2:00 AM"

monitoring:
  enabled: true
  tools: [prometheus, grafana]
  alerts:
    cpu_threshold: 80
    memory_threshold: 85
    disk_threshold: 90
```

✅ YAML Validation & Hands-On Practice

🔍 "Validate before you deploy - catch errors early!"

🌐 YAML Validation Methods

🔍 **Online Validators**

yamllint.com
jsonformatter.org/yaml-validator

✅ Quick & Free

🔧 **Code Editors**

VS Code, Sublime Text
Real-time validation

✅ Live Feedback

🐍 **Python Scripts**

Custom validation
Automated checking

✅ Programmable

🐍 Simple Python YAML Validator

📝 yaml_checker.py - Basic YAML Validation

```
#!/usr/bin/env python3
"""
Simple YAML Validation Script
Checks YAML files for syntax errors and basic structure
"""

import yaml
import sys
from pathlib import Path

def validate_yaml_file(file_path):
    """Validate a single YAML file"""
    try:
        print(f"🔍 Checking: {file_path}")

        # Check if file exists
        if not Path(file_path).exists():
            print(f"❌ File not found: {file_path}")
            return False

        # Read and parse YAML
        with open(file_path, 'r') as file:
            data = yaml.safe_load(file)

        # Basic validation checks
        if data is None:
            print(f"⚠️  {file_path}: File is empty or contains only comments")
            return True

        # Check data type
        if isinstance(data, dict):
            print(f"✅ {file_path}: Valid YAML dictionary with {len(data)} keys")
        elif isinstance(data, list):
            print(f"✅ {file_path}: Valid YAML list with {len(data)} items")
        else:
            print(f"✅ {file_path}: Valid YAML with {type(data).__name__} data")
        return True

    except yaml.YAMLError as e:
        print(f"❌ {file_path}: YAML syntax error")
        print(f"   Error details: {e}")
        return False
    except Exception as e:
        print(f"❌ {file_path}: Unexpected error - {e}")
        return False

def main():
    """Main validation function"""
    if len(sys.argv) < 2:
        print("Usage: python yaml_checker.py  [yaml_file2] ...")
        print("Example: python yaml_checker.py config.yml team.yml")
        sys.exit(1)

    print("🔍 YAML VALIDATION REPORT")
    print("=" * 40)

    all_valid = True

    # Validate each file
    for file_path in sys.argv[1:]:
        if not validate_yaml_file(file_path):
            all_valid = False
        print()  # Empty line between files

    # Final result
    if all_valid:
        print("🎉 All YAML files are valid!")
    else:
        print("⚠️  Some YAML files have errors - please fix them")
        sys.exit(1)

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

🎯 Hands-On YAML Practice Exercises

📝 Exercise 1: Fix the Broken YAML

```
# broken.yml - Find and fix 5+ errors
name:John Doe
age: 30
address:
street: 123 Main St
  city: Boston
    zip: 02101
hobbies:
- reading
  - swimming
- cooking
is_student:yes
grades: [A, B+, A-]
contact:
email:john@email.com
  phone: 555-1234
```

Tasks:

- Fix spacing after colons
- Correct indentation issues
- Align list items properly

🏗️ Exercise 2: Build a Configuration

Create a YAML file for a simple web application with:

- Application name and version
- Server settings (host, port)
- Database configuration
- List of enabled features
- Environment variables
- Logging settings

Validation Steps:

1. Save as `webapp.yml`
2. Use Python validator script
3. Test with online validator
4. Fix any errors found

- Validate with online tool

🚀 Exercise 3: Create a CI/CD Configuration

Build a simple GitHub Actions workflow YAML that:

- Triggers on push to main branch
- Runs on Ubuntu latest
- Has a job called "test"
- Checks out code
- Sets up Node.js version 18
- Installs dependencies
- Runs tests
- Uses environment variables
- Includes proper indentation
- Validates without errors

💡 **Bonus Challenge:** Add comments explaining each section and create a second job for building the application!

Prepared by: Rashi Rana
(Corporate Trainer)

🔍 YAML Linting vs Validation: Quality Control

🛡️ "Two layers of YAML quality assurance"

🤔 Linting vs Validation: What's the Difference?

🔍 YAML Linting

Purpose: Check style, formatting, and best practices

- **Indentation consistency** - 2 vs 4 spaces
- **Line length limits** - max 80/120 characters
- **Trailing whitespace** - remove extra spaces
- **Empty lines** - consistent spacing
- **Comments formatting** - proper placement
- **Key ordering** - alphabetical sorting
- **Quote consistency** - single vs double quotes

💡 **Think of it as:**
Grammar and style checker for YAML

✅ YAML Validation

Purpose: Check syntax correctness and structure

- **Syntax errors** - malformed YAML
- **Indentation errors** - incorrect nesting
- **Data type issues** - invalid values
- **Missing colons/spaces** - structural problems
- **Unclosed quotes** - string formatting
- **Invalid characters** - encoding issues
- **Schema compliance** - required fields

💡 **Think of it as:**
Spell checker and syntax verifier for YAML

🛠️ Popular YAML Linting Tools

🐍 **yamllint**
Python-based linter
Highly configurable
CI/CD integration
pip install yamllint

🌐 **Online Linters**
yamllint.com
codebeautify.org
Quick validation
No installation needed

🔧 **Editor Plugins**
VS Code YAML
Sublime Text
Real-time feedback
Live error highlighting

💻 Practical Linting Examples

🚫 Before Linting (Issues):

```
# Poor formatting and style issues
name:John Doe
```

✅ After Linting (Clean):

```
# Personal information
name: "John Doe"
```

```
age:  30
address:
    street: 123 Main St
  city: Boston
      zip: 02101

# Inconsistent quotes and spacing
hobbies: ['reading',"swimming", cooking]

# Long line exceeding limits
description: "This is a very long
description that exceeds the recommended
line length limit and should be broken
into multiple lines for better
readability"

# Missing comments and poor structure
database_config:
host: localhost
port: 5432
username: admin
```

```
age: 30
address:
  street: "123 Main St"
  city: "Boston"
  zip: "02101"

# Hobbies and interests
hobbies:
  - "reading"
  - "swimming"
  - "cooking"

# Description with proper line breaks
description: >
  This is a very long description that
has been
  properly formatted with line breaks for
better
  readability and maintainability.

# Database configuration
database_config:
  host: "localhost"
  port: 5432
  username: "admin"
```

🔧 Common yamllint Commands:

```
# Basic linting
yamllint myfile.yml

# Lint multiple files
yamllint *.yml

# Lint with specific config
yamllint -c .yamllint.yml myfile.yml

# Output in different formats
yamllint -f parsable myfile.yml
yamllint -f github-actions myfile.yml

# Lint entire directory
yamllint /path/to/yaml/files/

# Show only errors (ignore warnings)
yamllint -d "{extends: default, rules: {line-length: disable}}" myfile.yml
```

⚙️ Sample .yamllint.yml Configuration:

```
# .yamllint.yml - Custom linting rules
extends: default

rules:
  # Line length settings
  line-length:
    max: 120
    level: warning

  # Indentation rules
  indentation:
    spaces: 2
    indent-sequences: true

  # Comments formatting
  comments:
    min-spaces-from-content: 1

  # Disable some rules
  truthy: disable
  document-start: disable
```

🎯 YAML Linting Hands-On Workshop

🛠️ **"Practice makes perfect - Let's lint some YAML!"**

🚀 Workshop Setup

📦 Install yamllint:

```
# Install yamllint
pip install yamllint

# Verify installation
yamllint --version

# Get help
yamllint --help
```

💡 **Alternative:**
Use online linters if you can't install yamllint

📁 Create Workshop Files:

```
# Create workshop directory
mkdir yaml-linting-workshop
cd yaml-linting-workshop

# Create sample files
touch config.yml
touch docker-compose.yml
touch .yamllint.yml
```

📝 **Note:**
We'll create content for these files in the exercises

🔧 Exercise 1: Fix Common Linting Issues

📝 Create messy-config.yml with intentional issues:

```
# messy-config.yml - Contains multiple linting issues
---
app_name:MyApp
version: 1.0.0

# Server configuration
server:
host: 0.0.0.0
  port:   8080
    timeout: 30

database:
  type:postgresql
  host:localhost
  port: 5432
  credentials:
      username: admin
    password: secret123

# Features list with inconsistent formatting
features: [user_auth,email_notifications, analytics,reporting]

# Long description line that exceeds the recommended maximum line length and should be broken down
into multiple lines for better readability
description: "This is an extremely long description that violates the line length rule and should
be reformatted to improve readability and maintainability of the YAML file and make it easier to
work with"
```

```
# Inconsistent boolean values
debug_mode: yes
production_ready:true
maintenance: false

# Trailing spaces and empty lines issues


logging:
  level: INFO
  file: /var/log/app.log

# Mixed quotes and escaping issues
message: 'Don't use mixed quotes'
path: "C:\Windows\System32"
regex: "[a-zA-Z0-9]+"
```

🔍 Run yamllint and fix issues:

```
# Check for linting issues
yamllint messy-config.yml

# Expected issues to find and fix:
# 1. Missing space after colons (:)
# 2. Inconsistent indentation (2 vs 4 spaces)
# 3. Line too long (>80 characters)
# 4. Trailing whitespace
# 5. Too many blank lines
# 6. Inconsistent boolean values (yes/true/false)
# 7. Inconsistent list formatting
# 8. Mixed quote styles
# 9. Tab characters instead of spaces
```

🎯 Challenge: Fix These Broken YAML Files

📄 broken-ci.yml (CI/CD Pipeline with errors):

```
# broken-ci.yml - Fix this CI/CD configuration
name:Build and Deploy
on:
push:
branches:[main,develop]
  pull_request:
    branches: [ main ]

jobs:
build:
runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v3
      - name: Setup Node.js
        uses:actions/setup-node@v3
        with:
node-version: '18'
      - name:Install dependencies
        run: |
npm install
          npm run build
      - name: Run tests
        run: npm test
        env:
          NODE_ENV:production
          API_KEY: ${{ secrets.API_KEY }}

deploy:
    needs:build
    runs-on: ubuntu-latest
    if: github.ref == 'refs/heads/main'
    steps:
```

```
    - name: Deploy to production
      run: echo "Deploying..."
```

broken-k8s.yml (Kubernetes manifest with errors):

```
# broken-k8s.yml - Fix this Kubernetes deployment
apiVersion:apps/v1
kind:Deployment
metadata:
name:web-app
  namespace: default
spec:
replicas:3
  selector:
matchLabels:
      app: web-app
template:
    metadata:
      labels:
app:web-app
    spec:
      containers:
      - name:web-container
        image:nginx:1.21
        ports:
        - containerPort:80
        env:
        - name: ENV
value: production
        - name:DEBUG
          value: "false"
        resources:
          requests:
memory:"128Mi"
            cpu: "100m"
          limits:
            memory: "256Mi"
cpu:"200m"
---
apiVersion: v1
kind:Service
metadata:
  name: web-service
spec:
selector:
    app:web-app
  ports:
  - port:80
    targetPort: 80
  type:LoadBalancer
```

broken-ansible.yml (Ansible playbook with errors):

```
# broken-ansible.yml - Fix this Ansible playbook
---
- name:Deploy Web Application
hosts:webservers
  become:yes
  vars:
    app_name:myapp
    app_version: 1.0.0
    app_port:8080

  tasks:
  - name: Update package cache
    apt:
update_cache: yes
      cache_valid_time:3600

  - name:Install required packages
    apt:
      name: ['nginx', 'python3', 'python3-pip']
state:present

  - name: Create application directory
```

```
    file:
      path:/opt/{{ app_name }}
      state: directory
owner:www-data
      group: www-data
      mode:'0755'

  - name:Copy application files
    copy:
      src: ./app/
      dest: /opt/{{ app_name }}/
      owner: www-data
      group:www-data
      mode: '0644'
    notify:restart nginx

  - name: Template nginx configuration
    template:
src:nginx.conf.j2
      dest: /etc/nginx/sites-available/{{ app_name }}
    notify:
      - restart nginx

  handlers:
  - name:restart nginx
    service:
      name:nginx
      state: restarted
```

🎯 **Workshop Instructions:**

1. Copy each broken YAML file to your workshop directory
2. Run `yamllint filename.yml` on each file
3. Identify and fix all linting errors
4. Re-run yamllint until you get zero errors
5. Compare your fixes with teammates
6. **Bonus:** Create a custom .yamllint.yml config for each file type

⚙️ Exercise 2: Create Custom Linting Configuration

📝 Create .yamllint.yml with custom rules:

```
# .yamllint.yml - Custom linting configuration
extends: default

rules:
  # Line length - allow longer lines for URLs
  line-length:
    max: 120
    level: warning
    allow-non-breakable-words: true
    allow-non-breakable-inline-mappings: true

  # Indentation - enforce 2 spaces
  indentation:
    spaces: 2
    indent-sequences: true
    check-multi-line-strings: false

  # Comments - require space after #
  comments:
    min-spaces-from-content: 1
    require-starting-space: true

  # Empty lines - control blank line usage
  empty-lines:
    max: 2
    max-start: 0
    max-end: 1

  # Brackets - consistent spacing
  brackets:
    min-spaces-inside: 0
    max-spaces-inside: 1

  # Braces - consistent spacing
  braces:
    min-spaces-inside: 0
    max-spaces-inside: 1

  # Disable some strict rules for flexibility
  truthy:
    allowed-values: ['true', 'false', 'yes', 'no']
    check-keys: false

  document-start: disable
  document-end: disable
```

🧪 Test your configuration:

```
# Test with custom config
yamllint -c .yamllint.yml messy-config.yml

# Create a test file to verify rules
cat > test-rules.yml << 'EOF'
# Test file for custom rules
name: "Test App"
version: 1.0.0

# This line is intentionally very long to test the line-length rule configuration and see how it behaves with our custom settings

config:
  enabled: yes  # Test truthy rule
  debug: true   # Test truthy rule
  items: [ 1, 2, 3 ]  # Test brackets rule
  settings: { key: value }  # Test braces rule
EOF

# Test the rules
yamllint -c .yamllint.yml test-rules.yml
```

🎯 Challenge Exercise: Docker Compose Linting

Create a docker-compose.yml file and apply linting rules:

```
# Create docker-compose.yml with intentional issues
cat > docker-compose.yml << 'EOF'
version:'3.8'
services:
web:
image:nginx:latest
  ports:
- "80:80"
    - "443:443"
  environment:
    - ENV=production
      - DEBUG=false
  volumes:
      - ./html:/usr/share/nginx/html
database:
  image: postgres:13
    environment:
      POSTGRES_DB:myapp
      POSTGRES_USER: admin
      POSTGRES_PASSWORD:secret123
    volumes:
- postgres_data:/var/lib/postgresql/data
volumes:
postgres_data:
EOF

# Lint and fix the Docker Compose file
yamllint -c .yamllint.yml docker-compose.yml
```

🏆 **Success Criteria:**

All files should pass yamllint with zero errors and minimal warnings using your custom configuration.

Prepared by: Rashi Rana
(Corporate Trainer)

✅ YAML Linting Solutions & Best Practices

🎯 Correct YAML Examples & Solutions

✅ Corrected YAML Files

📝 clean-config.yml (Fixed version):

```yaml
# clean-config.yml - All linting issues fixed
---
app_name: MyApp
version: 1.0.0

# Server configuration
server:
  host: 0.0.0.0
  port: 8080
  timeout: 30

database:
  type: postgresql
  host: localhost
  port: 5432
  credentials:
    username: admin
    password: secret123

# Features list with consistent formatting
features:
  - user_auth
  - email_notifications
  - analytics
  - reporting

# Properly formatted long description
description: >
  This is a properly formatted long description that follows
  YAML best practices for readability and maintainability.
  It uses the folded scalar style for better line management.

# Consistent boolean values
debug_mode: true
production_ready: true
maintenance: false

logging:
  level: INFO
  file: /var/log/app.log

# Properly quoted strings
message: "Don't use mixed quotes"
path: "C:\\Windows\\System32"
regex: "[a-zA-Z0-9]+"
```

📝 fixed-ci.yml (Corrected CI/CD Pipeline):

```yaml
# fixed-ci.yml - Clean CI/CD configuration
name: Build and Deploy
```

```
on:
  push:
    branches: [main, develop]
  pull_request:
    branches: [main]

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Setup Node.js
        uses: actions/setup-node@v3
        with:
          node-version: '18'
      - name: Install dependencies
        run: |
          npm install
          npm run build
      - name: Run tests
        run: npm test
        env:
          NODE_ENV: production
          API_KEY: ${{ secrets.API_KEY }}

  deploy:
    needs: build
    runs-on: ubuntu-latest
    if: github.ref == 'refs/heads/main'
    steps:
      - name: Deploy to production
        run: echo "Deploying..."
```

📝 fixed-k8s.yml (Corrected Kubernetes manifest):

```
# fixed-k8s.yml - Clean Kubernetes deployment
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-app
  namespace: default
spec:
  replicas: 3
  selector:
    matchLabels:
      app: web-app
  template:
    metadata:
      labels:
        app: web-app
    spec:
      containers:
        - name: web-container
          image: nginx:1.21
          ports:
            - containerPort: 80
          env:
            - name: ENV
              value: production
            - name: DEBUG
              value: "false"
          resources:
            requests:
              memory: "128Mi"
              cpu: "100m"
            limits:
              memory: "256Mi"
              cpu: "200m"
---
apiVersion: v1
kind: Service
```

```
metadata:
  name: web-service
spec:
  selector:
    app: web-app
  ports:
    - port: 80
      targetPort: 80
  type: LoadBalancer
```

🏆 YAML Linting Best Practices

✅ Do's

- Use consistent indentation (2 or 4 spaces)
- Add space after colons (:)
- Use consistent boolean values
- Keep lines under 80-120 characters
- Use meaningful comments
- Quote strings when necessary
- Use block scalars for long text
- Validate syntax before committing

❌ Don'ts

- Mix tabs and spaces
- Leave trailing whitespace
- Use inconsistent indentation
- Mix quote styles unnecessarily
- Create overly long lines
- Use too many blank lines
- Ignore linting warnings
- Skip validation in CI/CD

🛠️ Recommended Linting Workflow

```
# 1. Install yamllint in your project
pip install yamllint

# 2. Create project-specific configuration
cat > .yamllint.yml << 'EOF'
extends: default
rules:
  line-length:
    max: 120
  indentation:
    spaces: 2
  truthy:
    allowed-values: ['true', 'false']
EOF

# 3. Add to pre-commit hooks
# .pre-commit-config.yaml
repos:
  - repo: https://github.com/adrienverge/yamllint
    rev: v1.32.0
    hooks:
      - id: yamllint

# 4. Integrate with CI/CD
# In your CI pipeline:
yamllint .
# Or for specific files:
yamllint config/ *.yml *.yaml
```

Prepared by: Rashi Rana
(Corporate Trainer)