Linux-ShellScripting-Bash

K19G

2026-02-28

Table of contents

Preface

Welcome to the comprehensive guide on Linux Shell Scripting and Bash Programming. This book is designed to take you from a complete beginner to an advanced shell scripter, covering everything from basic shell concepts to complex automation scripts.

What You'll Learn

- Understanding different types of shells and their purposes
- Fundamentals of shell scripting and script execution
- Comprehensive Bash programming techniques
- Variables, arrays, and data manipulation
- Control structures and flow control
- Functions and modular programming
- String manipulation and pattern matching
- File operations and system interaction
- Process management and job control
- Error handling and debugging strategies
- Advanced Bash features and best practices
- Real-world automation examples

Who This Book Is For

This book is perfect for: - System administrators looking to automate tasks - Developers wanting to improve their command-line skills - Students learning Linux and Unix systems - Anyone interested in shell scripting and automation

How to Use This Book

You can read this book in various formats: - Online HTML version with interactive examples - Downloadable PDF for offline reading - EPUB for e-readers

Each chapter builds upon the previous ones, so it's recommended to read them in order if you're new to shell scripting.

Prerequisites

- Basic familiarity with Linux/Unix command line
- Understanding of file systems and basic commands
- No prior programming experience required

Acknowledgments

This book is dedicated to the open-source community and all the developers who have contributed to making Linux and Bash the powerful tools they are today.

Let's begin your journey into the powerful world of shell scripting!

Intro To Shells

Shell vs Terminal: Understanding the Difference

Many people use the terms "shell" and "terminal" interchangeably, but they are actually different components that work together to provide your command-line experience.

What is a Terminal?

A terminal (also called terminal emulator) is a program that provides a window for you to interact with the shell. It handles:

- **Display**: Shows text output from programs
- **Input**: Captures keyboard input and sends it to the shell
- **Window Management**: Provides scrolling, resizing, and visual features
- **Character Encoding**: Handles text rendering and fonts

Common Terminal Emulators

Linux:

- **GNOME Terminal**: Default on GNOME desktop
- **Konsole**: KDE's terminal emulator
- **xterm**: Classic X11 terminal
- **Alacritty**: GPU-accelerated terminal
- **Terminator**: Terminal with split panes

macOS:

- **Terminal.app**: Built-in terminal
- **iTerm2**: Popular third-party option

Windows:

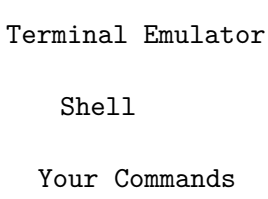
- **Windows Terminal**: Modern terminal for Windows 10/11
- **Command Prompt**: Traditional Windows terminal
- **PowerShell**: Microsoft's advanced shell and terminal

What is a Shell?

A shell is the command interpreter that runs inside the terminal. It:

- **Processes Commands**: Interprets what you type
- **Manages Environment**: Handles variables and settings
- **Executes Programs**: Launches other applications
- **Provides Scripting**: Enables automation through scripts

The Relationship



Practical Examples

Example 1: Multiple Shells in One Terminal

```
# Start in bash
$ echo $SHELL
/bin/bash

# Switch to zsh
$ zsh
$ echo $SHELL
/bin/zsh

# Switch to fish
$ fish
$ echo $SHELL
/usr/bin/fish

# Exit back to previous shells
$ exit  # Back to zsh
$ exit  # Back to bash
```

Example 2: Same Shell in Different Terminals

You can run the same shell (like bash) in multiple terminal windows simultaneously. Each terminal window is independent, but they're all running the same shell program.

Example 3: Remote Shells

```
# Connect to remote server via SSH
$ ssh user@remote-server
# Now you're running a shell on the remote machine
# but displaying it in your local terminal
```

Terminal Features vs Shell Features

Terminal Features:

- **Colors and Themes**: Visual appearance
- **Font Settings**: Text rendering
- **Window Management**: Tabs, splits, transparency
- **Copy/Paste**: Clipboard integration
- **Scrollback**: History of displayed text

Shell Features:

- **Command History**: Previously executed commands
- **Tab Completion**: Auto-complete functionality
- **Variables**: Environment and user-defined variables
- **Scripting**: Programming capabilities
- **Job Control**: Managing background processes

Configuration Files

Terminal Configuration:

- Usually GUI-based settings
- Profiles for different appearances
- Keyboard shortcuts

Shell Configuration:

```
# Bash configuration files
~/.bashrc      # Interactive non-login shells
~/.bash_profile # Login shells
~/.bash_logout  # Logout cleanup

# Zsh configuration files
~/.zshrc       # Main configuration
~/.zprofile    # Login shells

# Fish configuration
~/.config/fish/config.fish
```

Choosing Terminal and Shell Combinations

For Development:

- **Terminal**: iTerm2 (macOS) or GNOME Terminal (Linux)
- **Shell**: bash or zsh with customizations

For System Administration:

- **Terminal**: Any reliable terminal emulator
- **Shell**: bash for compatibility

For Beginners:

- **Terminal**: Default system terminal
- **Shell**: bash (most common) or fish (user-friendly)

Common Misconceptions

Misconception 1: "I'm using Terminal"

Reality: You're using a terminal emulator to access a shell.

Misconception 2: "Bash is a terminal"

Reality: Bash is a shell that runs inside a terminal.

Misconception 3: "All terminals are the same"

Reality: Different terminals have different features and capabilities.

Practical Tips

Check what terminal you're using:

```
$ echo $TERM
xterm-256color

$ ps -o comm= -p "$PPID"
gnome-terminal-
```

Check what shell you're using:

```
$ echo $0
bash

$ ps -p $$
  PID TTY          TIME CMD
 1234 pts/0    00:00:00 bash
```

Launch different shells:

```
# Start a new shell session
$ bash
$ zsh
$ fish

# Run a single command in a different shell
$ zsh -c "echo 'This runs in zsh'"
$ fish -c "echo 'This runs in fish'"
```

Summary

- **Terminal**: The window/interface you see and interact with
- **Shell**: The command interpreter that processes your commands
- **Relationship**: The terminal displays the shell's input/output
- **Independence**: You can use different shells in the same terminal, or the same shell in different terminals

Understanding this distinction helps you troubleshoot issues, customize your environment effectively, and communicate more precisely about command-line tools.

Types of Shells

Linux and Unix systems support various types of shells, each with its own features and capabilities. Understanding the different shells helps you choose the right tool for your needs.

Common Shell Types

1. Bourne Shell (sh)

- **Original**: The first Unix shell, developed by Stephen Bourne
- **Features**: Basic scripting capabilities, simple syntax
- **Usage**: Still used for system scripts and compatibility
- **Location**: Usually `/bin/sh`

```
# Check if sh is available
$ which sh
/bin/sh
```

2. Bash (Bourne Again Shell)

- **Most Popular**: Default shell on most Linux distributions
- **Features**: Enhanced version of Bourne shell with many improvements
- **Advantages**:
  - Command history
  - Tab completion
  - Job control
  - Arrays and associative arrays
  - Advanced scripting features

```
# Check Bash version
$ bash --version
GNU bash, version 5.1.16(1)-release
```

3. Zsh (Z Shell)

- **Advanced**: Feature-rich shell with powerful customization
- **Features**:
  - Excellent tab completion
  - Spelling correction
  - Plugin system
  - Themes support
- **Popular Framework**: Oh My Zsh

```
# Install zsh (Ubuntu/Debian)
$ sudo apt install zsh

# Make zsh default shell
$ chsh -s $(which zsh)
```

4. Fish (Friendly Interactive Shell)

- **User-Friendly**: Designed for ease of use
- **Features**:
  - Syntax highlighting
  - Auto-suggestions
  - Web-based configuration
  - No configuration required

```
# Install fish (Ubuntu/Debian)
$ sudo apt install fish

# Start fish shell
$ fish
```

5. Dash (Debian Almquist Shell)

- **Lightweight**: Minimal shell focused on speed
- **Usage**: Often used as `/bin/sh` on Debian-based systems
- **Purpose**: System scripts and boot processes

6. Tcsh (TENEX C Shell)

- **C-like Syntax**: Similar to C programming language
- **Features**: Command history, job control, aliases
- **Usage**: Less common in modern systems

7. Ksh (Korn Shell)

- **Commercial**: Developed by David Korn at Bell Labs
- **Features**: Combines features of Bourne and C shells
- **Variants**: ksh88, ksh93, mksh (MirBSD Korn Shell)

Checking Available Shells

View all available shells:

```
$ cat /etc/shells
/bin/sh
/bin/bash
/usr/bin/bash
/bin/rbash
/usr/bin/rbash
/bin/dash
/usr/bin/dash
/usr/bin/zsh
/usr/bin/fish
```

Check your current shell:

```
$ echo $SHELL
/bin/bash

# Or use ps command
$ ps -p $$
  PID TTY          TIME CMD
 1234 pts/0    00:00:00 bash
```

Check shell being used by a script:

```
$ ps -o pid,ppid,cmd
```

Shell Comparison

Feature	sh	bash	zsh	fish	dash
Speed	Fast	Medium	Medium	Medium	Very Fast
Scripting	Basic	Advanced	Advanced	Good	Basic

Feature	sh	bash	zsh	fish	dash
Interactive	Basic	Good	Excellent	Excellent	Basic
Compatibility	High	High	Good	Low	High
Customization	Limited	Good	Excellent	Good	Limited

Choosing the Right Shell

For Scripting:

- **bash**: Best balance of features and compatibility
- **sh**: Maximum portability across systems
- **dash**: When speed is critical

For Interactive Use:

- **zsh**: Power users who want customization
- **fish**: Beginners who want user-friendly features
- **bash**: Good default choice for most users

For System Administration:

- **bash**: Most common, good documentation
- **sh**: For system scripts that need portability

Switching Between Shells

Temporarily switch:

```
# Start a new shell session
$ zsh
$ fish
$ bash

# Exit back to previous shell
$ exit
```

Change default shell:

```
# Change default shell for current user
$ chsh -s /usr/bin/zsh

# Verify the change
$ grep $USER /etc/passwd
```

Shell Compatibility

When writing scripts, consider compatibility:

```
#!/bin/bash
# This script requires bash-specific features

#!/bin/sh
# This script should work with any POSIX shell

#!/usr/bin/env bash
# Portable way to find bash
```

Understanding the different types of shells available helps you make informed decisions about which shell to use for different tasks and environments.

What is a Shell?

A shell is a command-line interface that acts as an intermediary between you and the operating system kernel. It's called a "shell" because it wraps around the kernel, providing a user-friendly way to interact with the system.

The Role of a Shell

The shell serves several critical functions:

1. **Command Interpreter**: Reads and executes commands you type
2. **Program Launcher**: Starts other programs and applications
3. **Environment Manager**: Manages environment variables and settings
4. **Scripting Platform**: Allows automation through script files

How Shells Work

When you type a command, here's what happens:

```
$ ls -la
```

1. The shell reads your input
2. Parses the command and arguments
3. Locates the program (`ls`) in the system PATH
4. Executes the program with the specified arguments (`-la`)
5. Displays the output back to you

Interactive vs Non-Interactive Shells

Interactive Shell

- Accepts commands from user input
- Provides a prompt (like `$` or `#`)
- Offers features like command history and tab completion

Non-Interactive Shell

- Executes scripts without user interaction
- Used for automation and batch processing
- No prompt or interactive features

Login vs Non-Login Shells

Login Shell

- Started when you log into the system
- Reads configuration files like `.bash_profile`
- Sets up the initial environment

Non-Login Shell

- Started from within another shell or application
- Reads different configuration files like `.bashrc`
- Inherits environment from parent shell

Shell Capabilities

Modern shells provide powerful features:

- **Command History**: Access previously executed commands
- **Tab Completion**: Auto-complete commands and filenames
- **Wildcards**: Use patterns like `*.txt` to match multiple files
- **Pipes and Redirection**: Chain commands and redirect output
- **Variables**: Store and manipulate data
- **Control Structures**: Conditional statements and loops
- **Functions**: Create reusable code blocks

Example: Basic Shell Interaction

```
# Display current directory
$ pwd
/home/username

# List files
$ ls
Documents  Downloads  Pictures
```

```
# Create a directory
$ mkdir my_scripts

# Navigate to the directory
$ cd my_scripts

# Check current location
$ pwd
/home/username/my_scripts
```

Understanding what a shell is and how it works is fundamental to becoming proficient in shell scripting and system administration.

Shell Scripting

Your First Shell Script

Let's create your first shell script step by step. This hands-on approach will help you understand the basics and get you started with shell scripting.

Step 1: Create Your First Script

Using a Text Editor

Open your favorite text editor and create a new file called `hello.sh`:

```
# Using nano
$ nano hello.sh

# Using vim
$ vim hello.sh

# Using VS Code
$ code hello.sh
```

Write the Script

Type the following content into your file:

```
#!/bin/bash
# My first shell script
# Purpose: Display a greeting message

echo "Hello, World!"
echo "Welcome to shell scripting!"
echo "Today's date is: $(date)"
echo "Current user: $(whoami)"
echo "Current directory: $(pwd)"
```

Save the File

- **nano**: Press `Ctrl+X`, then `Y`, then `Enter`
- **vim**: Press `Esc`, type `:wq`, then `Enter`
- **VS Code**: Press `Ctrl+S`

Step 2: Understanding the Script

Let's break down each part:

Shebang Line

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

- **Purpose**: Tells the system to use bash to execute this script
- **Must be**: The very first line of the script
- **Alternative**: `#!/bin/sh` for POSIX compatibility

Comments

```
# My first shell script
# Purpose: Display a greeting message
```

- **Purpose**: Document what the script does
- **Syntax**: Lines starting with `#` (except shebang)
- **Best Practice**: Always comment your scripts

Commands

```
echo "Hello, World!"
```

- **echo**: Command to display text
- **Quotes**: Protect text with spaces and special characters

Command Substitution

```
echo "Today's date is: $(date)"
```

- **$(command)**: Executes command and substitutes its output
- **Alternative**: Backticks `` `date` `` (older syntax)

Step 3: Make the Script Executable

Before you can run the script, you need to make it executable:

```
# Check current permissions
$ ls -l hello.sh
```

```
-rw-r--r-- 1 user user 245 Jan 15 10:30 hello.sh

# Make it executable
$ chmod +x hello.sh

# Verify permissions changed
$ ls -l hello.sh
-rwxr-xr-x 1 user user 245 Jan 15 10:30 hello.sh
```

Understanding Permissions

- **r**: Read permission
- **w**: Write permission
- **x**: Execute permission
- **First rwx**: Owner permissions
- **Second rwx**: Group permissions
- **Third rwx**: Other users permissions

Step 4: Run Your Script

There are several ways to execute your script:

Method 1: Direct Execution

```
$ ./hello.sh
Hello, World!
Welcome to shell scripting!
Today's date is: Mon Jan 15 10:35:22 EST 2024
Current user: john
Current directory: /home/john/scripts
```

Method 2: Using bash Command

```
$ bash hello.sh
# Same output as above
```

Method 3: Using sh Command

```
$ sh hello.sh
# Same output as above
```

Method 4: Full Path

```
$ /home/john/scripts/hello.sh
# Same output as above
```

Step 5: Enhance Your Script

Let's make the script more interactive:

```
#!/bin/bash
# Enhanced greeting script
# Purpose: Interactive greeting with user input

echo "===================================="
echo "    Welcome to Shell Scripting!"
echo "===================================="
echo

# Get user's name
echo "What's your name?"
read name

# Get user's favorite color
echo "What's your favorite color?"
read color

# Display personalized greeting
echo
echo "Hello, $name!"
echo "I see your favorite color is $color."
echo "That's a great choice!"
echo
echo "System Information:"
echo "==================="
echo "Date: $(date)"
echo "User: $(whoami)"
echo "Home Directory: $HOME"
echo "Current Directory: $(pwd)"
```

```
echo "Shell: $SHELL"
echo
echo "Have a great day, $name!"
```

Save this as `enhanced_hello.sh` and run it:

```
$ chmod +x enhanced_hello.sh
$ ./enhanced_hello.sh
```

Step 6: Adding Error Handling

Let's add some basic error handling:

```
#!/bin/bash
# Script with error handling
# Purpose: Demonstrate basic error handling

set -e  # Exit immediately if a command exits with a non-zero status

echo "Starting script..."

# Check if a directory exists
if [ -d "/home" ]; then
    echo "  /home directory exists"
else
    echo "  /home directory not found"
    exit 1
fi

# Try to create a test file
TEST_FILE="test_file.txt"
echo "Creating test file: $TEST_FILE"

if echo "This is a test" > "$TEST_FILE"; then
    echo "  Test file created successfully"

    # Clean up
    rm "$TEST_FILE"
    echo "  Test file cleaned up"
else
    echo "  Failed to create test file"
    exit 1
fi

echo "Script completed successfully!"
```

Step 7: Script with Functions

Let's organize code using functions:

```
#!/bin/bash
# Script with functions
# Purpose: Demonstrate function usage

# Function to display a separator
show_separator() {
    echo "================================="
}

# Function to display system info
show_system_info() {
    echo "System Information:"
    echo "OS: $(uname -s)"
    echo "Kernel: $(uname -r)"
    echo "Architecture: $(uname -m)"
    echo "Hostname: $(hostname)"
}

# Function to display disk usage
show_disk_usage() {
    echo "Disk Usage:"
    df -h | head -5
}

# Main script execution
main() {
    show_separator
    echo "    System Status Report"
    show_separator
    echo

    show_system_info
    echo

    show_disk_usage
    echo

    show_separator
    echo "Report generated on: $(date)"
    show_separator
}

# Call main function
```

```
main
```

Common Beginner Mistakes

1. Forgetting the Shebang

```
# Wrong - no shebang
echo "Hello World"

# Correct
#!/bin/bash
echo "Hello World"
```

2. Not Making Script Executable

```
# This will fail
$ ./myscript.sh
bash: ./myscript.sh: Permission denied

# Fix it
$ chmod +x myscript.sh
$ ./myscript.sh
```

3. Wrong Path in Shebang

```
# Might not work on all systems
#!/usr/bin/bash

# More portable
#!/bin/bash

# Most portable
#!/usr/bin/env bash
```

4. Spaces in Variable Assignment

```
# Wrong - spaces around =
name = "John"

# Correct - no spaces
```

```
name="John"
```

Best Practices for Beginners

1. Always Use Shebang

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

2. Add Comments

```
# What the script does
# Author and date
# Usage instructions
```

3. Use Meaningful Names

```
# Good
backup_directory="/backup"
user_name="john"

# Avoid
dir="/backup"
n="john"
```

4. Quote Variables

```
# Good
echo "Hello, $name"
cp "$source_file" "$destination"

# Risky
echo Hello, $name
cp $source_file $destination
```

5. Check for Errors

```
#!/bin/bash
set -e  # Exit on error

# Or check individual commands
if ! command_that_might_fail; then
    echo "Command failed!"
    exit 1
fi
```

Testing Your Scripts

1. Syntax Check

```
$ bash -n myscript.sh
# No output means no syntax errors
```

2. Debug Mode

```
$ bash -x myscript.sh
# Shows each command as it's executed
```

3. Use shellcheck

```
$ shellcheck myscript.sh
# Provides suggestions for improvement
```

Next Steps

Now that you've created your first script, you can:

1. **Experiment**: Modify the examples and see what happens
2. **Practice**: Create scripts for tasks you do regularly
3. **Learn More**: Study variables, conditionals, and loops
4. **Read Others' Scripts**: Look at system scripts in `/etc/init.d/`
5. **Use Version Control**: Start tracking your scripts with git

Congratulations! You've written and executed your first shell script. This is the foundation for all the advanced scripting techniques you'll learn in the following chapters.

Script Execution Methods

Understanding how to execute shell scripts is crucial for effective shell scripting. There are multiple ways to run scripts, each with its own use cases and implications.

Execution Methods Overview

Method	Syntax	New Process	Environment	Use Case
Direct	`./script.sh`	Yes	Inherited	Most common
Interpreter	`bash script.sh`	Yes	Inherited	Testing/debugging
Source	`source script.sh`	No	Modified	Environment setup
Absolute Path	`/path/to/script.sh`	Yes	Inherited	System scripts

Method 1: Direct Execution

This is the most common way to run scripts.

Prerequisites

```
# Script must be executable
$ chmod +x myscript.sh

# Script must have proper shebang
$ head -1 myscript.sh
#!/bin/bash
```

Execution

```
# Run from current directory
$ ./myscript.sh

# Run from anywhere (if in PATH)
```

```
$ myscript.sh
```

Example Script

```
#!/bin/bash
# direct_example.sh
echo "Running in process: $$"
echo "Parent process: $PPID"
echo "Current directory: $(pwd)"
export NEW_VAR="Hello from script"
```

Test It

```
$ chmod +x direct_example.sh
$ ./direct_example.sh
Running in process: 12345
Parent process: 12344
Current directory: /home/user/scripts

# Check if NEW_VAR exists in current shell
$ echo $NEW_VAR
# (empty - variable doesn't exist in parent shell)
```

Method 2: Using Shell Interpreter

Run the script by explicitly calling the shell interpreter.

Bash Interpreter

```
$ bash myscript.sh
```

Other Interpreters

```
$ sh myscript.sh      # POSIX shell
$ zsh myscript.sh     # Z shell
$ dash myscript.sh    # Debian Almquist shell
```

Advantages

- **No execute permission needed**
- **Override shebang**: Force specific interpreter
- **Testing**: Test script with different shells

Example

```
# Create script without execute permission
$ cat > test_interpreter.sh << 'EOF'
#!/bin/bash
echo "Shell: $0"
echo "Bash version: $BASH_VERSION"
EOF

# Run without making executable
$ bash test_interpreter.sh
Shell: bash
Bash version: 5.1.16(1)-release

# Try with different shell
$ sh test_interpreter.sh
Shell: sh
Bash version:
```

Method 3: Source (Dot) Command

Execute script in the current shell environment.

Syntax

```
$ source myscript.sh
# or
$ . myscript.sh
```

Key Characteristics

- **No new process**: Runs in current shell
- **Environment changes persist**: Variables and functions remain
- **No shebang needed**: Uses current shell
- **No execute permission needed**

Example Script

```
# source_example.sh
echo "Setting up environment..."
export PROJECT_ROOT="/home/user/myproject"
export PATH="$PROJECT_ROOT/bin:$PATH"

# Define a function
greet() {
    echo "Hello from sourced function!"
}

echo "Environment setup complete"
echo "PROJECT_ROOT: $PROJECT_ROOT"
```

Test Sourcing

```
# Before sourcing
$ echo $PROJECT_ROOT
# (empty)

# Source the script
$ source source_example.sh
Setting up environment...
Environment setup complete
PROJECT_ROOT: /home/user/myproject

# After sourcing - variables persist
$ echo $PROJECT_ROOT
/home/user/myproject

# Function is available
$ greet
Hello from sourced function!
```

Method 4: Absolute Path Execution

Run script using its full path.

Examples

```
# Full path execution
$ /home/user/scripts/myscript.sh

# Using $HOME variable
$ $HOME/scripts/myscript.sh

# Using tilde expansion
$ ~/scripts/myscript.sh
```

Use Cases

- **System scripts**: `/etc/init.d/apache2 start`
- **Cron jobs**: Full paths prevent PATH issues
- **Remote execution**: SSH with full paths

Method 5: Command Substitution

Execute script and capture output.

Syntax

```
# Modern syntax
result=$(./myscript.sh)

# Legacy syntax (avoid)
result=`./myscript.sh`
```

Example

```
# Script that returns a value
$ cat > get_timestamp.sh << 'EOF'
#!/bin/bash
date +%Y%m%d_%H%M%S
EOF

$ chmod +x get_timestamp.sh

# Capture output
$ timestamp=$(./get_timestamp.sh)
```

```
$ echo "Backup file: backup_$timestamp.tar.gz"
Backup file: backup_20240115_143022.tar.gz
```

Execution Context and Environment

Process Creation

```
#!/bin/bash
# process_info.sh
echo "Script PID: $$"
echo "Parent PID: $PPID"
echo "Shell: $0"
ps -f --pid $$ --pid $PPID
```

Environment Inheritance

```
#!/bin/bash
# env_test.sh
echo "PATH: $PATH"
echo "HOME: $HOME"
echo "USER: $USER"
echo "Custom var: $MY_CUSTOM_VAR"
```

Test Environment

```
# Set a custom variable
$ export MY_CUSTOM_VAR="test value"

# Run script - inherits environment
$ ./env_test.sh
PATH: /usr/local/bin:/usr/bin:/bin
HOME: /home/user
USER: user
Custom var: test value
```

Script Arguments and Parameters

Passing Arguments

```
$ ./myscript.sh arg1 arg2 arg3
```

Accessing Arguments

```
#!/bin/bash
# args_example.sh
echo "Script name: $0"
echo "First argument: $1"
echo "Second argument: $2"
echo "All arguments: $@"
echo "Number of arguments: $#"
```

Test Arguments

```
$ ./args_example.sh hello world 123
Script name: ./args_example.sh
First argument: hello
Second argument: world
All arguments: hello world 123
Number of arguments: 3
```

Exit Status and Error Handling

Exit Status

```
#!/bin/bash
# exit_status.sh
echo "Performing operation..."

# Simulate success or failure
if [ "$1" = "fail" ]; then
    echo "Operation failed!"
    exit 1
else
    echo "Operation successful!"
    exit 0
fi
```

Check Exit Status

```
$ ./exit_status.sh success
Operation successful!
$ echo $?
0

$ ./exit_status.sh fail
Operation failed!
$ echo $?
1
```

Advanced Execution Techniques

Background Execution

```
# Run script in background
$ ./long_running_script.sh &
[1] 12345

# Check background jobs
$ jobs
[1]+  Running    ./long_running_script.sh &
```

Conditional Execution

```
# Run second command only if first succeeds
$ ./script1.sh && ./script2.sh

# Run second command only if first fails
$ ./script1.sh || ./script2.sh

# Always run second command
$ ./script1.sh; ./script2.sh
```

Piping Script Output

```
# Pipe script output to another command
$ ./generate_data.sh | sort | uniq

# Redirect output to file
$ ./report_script.sh > report.txt 2>&1
```

Debugging Script Execution

Debug Mode

```
# Show commands as they execute
$ bash -x myscript.sh

# Or add to script
#!/bin/bash -x
```

Verbose Mode

```
# Show input lines as they're read
$ bash -v myscript.sh
```

Syntax Check

```
# Check syntax without executing
$ bash -n myscript.sh
```

Combined Options

```
# Multiple debug options
$ bash -xv myscript.sh
```

Security Considerations

File Permissions

```
# Secure script permissions
$ chmod 755 myscript.sh  # Owner: rwx, Others: r-x
$ chmod 700 myscript.sh  # Owner only: rwx
```

PATH Security

```
#!/bin/bash
# Use absolute paths for security
/bin/ls /home/user
/usr/bin/whoami
```

Input Validation

```
#!/bin/bash
# Validate input before execution
if [ $# -ne 1 ]; then
    echo "Usage: $0 <filename>"
    exit 1
fi

filename="$1"
if [ ! -f "$filename" ]; then
    echo "Error: File '$filename' not found"
    exit 1
fi
```

Best Practices

1. Always Use Shebang

```
#!/bin/bash
# or
#!/usr/bin/env bash  # More portable
```

2. Set Error Handling

```
#!/bin/bash
set -euo pipefail  # Exit on error, undefined vars, pipe failures
```

3. Use Appropriate Execution Method

- **Direct execution**: General purpose scripts
- **Source**: Environment setup scripts
- **Interpreter**: Testing and debugging
- **Background**: Long-running processes

4. Handle Arguments Properly

```
#!/bin/bash
if [ $# -eq 0 ]; then
    echo "Usage: $0 <arguments>"
    exit 1
fi
```

5. Provide Clear Output

```
#!/bin/bash
echo "Starting process..."
# ... script logic ...
echo "Process completed successfully"
```

Understanding these execution methods gives you the flexibility to run scripts in the most appropriate way for your specific needs and environment.

What is Shell Scripting?

Shell scripting is the practice of writing programs (scripts) that are executed by a shell. These scripts automate tasks, combine multiple commands, and create powerful workflows that would be tedious to perform manually.

Definition and Purpose

A shell script is a text file containing a series of commands that the shell can execute. Instead of typing commands one by one, you write them in a file and execute the entire sequence at once.

Key Benefits:

- **Automation**: Eliminate repetitive manual tasks
- **Consistency**: Ensure tasks are performed the same way every time
- **Efficiency**: Save time and reduce human error
- **Scheduling**: Run scripts automatically at specific times
- **Complex Logic**: Implement decision-making and loops

When to Use Shell Scripts

Perfect for:

- **System Administration**: User management, backups, log rotation
- **File Operations**: Batch processing, file organization
- **Development Workflows**: Build processes, deployment scripts
- **Data Processing**: Log analysis, report generation
- **System Monitoring**: Health checks, alert systems

Not Ideal for:

- **Complex Calculations**: Use Python, R, or specialized tools
- **GUI Applications**: Shell scripts are command-line oriented
- **Performance-Critical Tasks**: Compiled languages are faster
- **Cross-Platform Applications**: Shell scripts are OS-specific

Types of Shell Scripts

1. Simple Command Sequences

```
#!/bin/bash
# Backup script
cp /home/user/documents/* /backup/
tar -czf /backup/backup_$(date +%Y%m%d).tar.gz /backup/*.txt
echo "Backup completed at $(date)"
```

2. Interactive Scripts

```
#!/bin/bash
# User input script
echo "What's your name?"
read name
echo "Hello, $name! Welcome to shell scripting."
```

3. System Administration Scripts

```
#!/bin/bash
# System monitoring script
echo "System Status Report - $(date)"
echo "============================"
echo "Disk Usage:"
df -h
echo "Memory Usage:"
free -h
echo "CPU Load:"
uptime
```

4. Data Processing Scripts

```
#!/bin/bash
# Log analysis script
echo "Analyzing web server logs..."
grep "ERROR" /var/log/apache2/error.log | wc -l
echo "Total errors found"
```

Shell Script Components

1. Shebang Line

```
#!/bin/bash
# Tells the system which interpreter to use
```

2. Comments

```
# This is a single-line comment
: '
This is a
multi-line comment
'
```

3. Variables

```
name="John"
age=25
echo "Name: $name, Age: $age"
```

4. Commands

```
ls -la
pwd
date
```

5. Control Structures

```
if [ $age -gt 18 ]; then
    echo "Adult"
else
    echo "Minor"
fi
```

Script Execution Methods

1. Make Executable and Run

```
# Make script executable
$ chmod +x myscript.sh

# Run the script
$ ./myscript.sh
```

2. Run with Shell Interpreter

```
# Run with bash
$ bash myscript.sh

# Run with sh
$ sh myscript.sh
```

3. Source the Script

```
# Execute in current shell (affects current environment)
$ source myscript.sh
# or
$ . myscript.sh
```

Script Structure Best Practices

Basic Template:

```
#!/bin/bash

# Script: example.sh
# Purpose: Demonstrate shell script structure
# Author: Your Name
# Date: $(date)

# Set strict error handling
set -euo pipefail

# Global variables
SCRIPT_DIR="$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd)"
```

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

# Functions
log_message() {
    echo "$(date '+%Y-%m-%d %H:%M:%S') - $1" | tee -a "$LOG_FILE"
}

main() {
    log_message "Script started"

    # Main script logic here
    echo "Hello, World!"

    log_message "Script completed"
}

# Execute main function
main "$@"
```

Common Shell Scripting Patterns

1. Error Handling

```
#!/bin/bash
set -e  # Exit on any error

command_that_might_fail || {
    echo "Command failed!"
    exit 1
}
```

2. Argument Processing

```
#!/bin/bash
if [ $# -eq 0 ]; then
    echo "Usage: $0 <filename>"
    exit 1
fi

filename="$1"
echo "Processing file: $filename"
```

3. Configuration Files

```
#!/bin/bash
# Load configuration
CONFIG_FILE="config.conf"
if [ -f "$CONFIG_FILE" ]; then
    source "$CONFIG_FILE"
else
    echo "Configuration file not found!"
    exit 1
fi
```

Shell Scripting vs Other Languages

Shell Scripts Excel At:

- System integration
- File manipulation
- Process management
- Quick automation tasks
- Gluing together existing tools

Other Languages Better For:

- **Python**: Complex logic, data analysis, web development
- **C/C++**: Performance-critical applications
- **JavaScript**: Web applications, Node.js servers
- **Go**: Network services, system tools

Development Environment

Essential Tools:

- **Text Editor**: vim, nano, VS Code, or any editor
- **Shell**: bash, zsh, or your preferred shell
- **Version Control**: git for tracking changes
- **Testing**: shellcheck for syntax checking

Useful Commands:

```
# Check script syntax
$ bash -n myscript.sh

# Debug script execution
$ bash -x myscript.sh

# Use shellcheck for best practices
$ shellcheck myscript.sh
```

Real-World Example

Here's a practical backup script:

```
#!/bin/bash
# backup.sh - Simple backup script

# Configuration
SOURCE_DIR="/home/user/documents"
BACKUP_DIR="/backup"
DATE=$(date +%Y%m%d_%H%M%S)
BACKUP_NAME="backup_$DATE.tar.gz"

# Create backup directory if it doesn't exist
mkdir -p "$BACKUP_DIR"

# Create backup
echo "Starting backup of $SOURCE_DIR..."
tar -czf "$BACKUP_DIR/$BACKUP_NAME" "$SOURCE_DIR"

# Check if backup was successful
if [ $? -eq 0 ]; then
    echo "Backup completed successfully: $BACKUP_DIR/$BACKUP_NAME"

    # Remove backups older than 7 days
    find "$BACKUP_DIR" -name "backup_*.tar.gz" -mtime +7 -delete
    echo "Old backups cleaned up"
else
    echo "Backup failed!"
    exit 1
fi
```

Shell scripting is a powerful skill that bridges the gap between simple command-line usage and full programming. It's an essential tool for anyone working with Linux or Unix systems.

Bash Intro

Bash Environment

The Bash environment consists of variables, settings, and configurations that determine how Bash behaves. Understanding and managing this environment is crucial for effective shell scripting and system administration.

Environment Variables

Environment variables are key-value pairs that affect how programs run. They're inherited by child processes and available to all programs launched from the shell.

Viewing Environment Variables

```
# Display all environment variables
$ env
PATH=/usr/local/bin:/usr/bin:/bin
HOME=/home/username
USER=username
SHELL=/bin/bash

# Display specific variable
$ echo $HOME
/home/username

# Alternative method
$ printenv HOME
/home/username

# List all variables (including shell variables)
$ set
```

Common Environment Variables

System Variables

```
# User information
echo $USER          # Current username
echo $HOME          # User's home directory
echo $UID           # User ID number
echo $GROUPS        # User's group memberships

# System information
echo $HOSTNAME      # System hostname
echo $HOSTTYPE      # Machine type
echo $OSTYPE        # Operating system type
echo $MACHTYPE      # Machine description

# Shell information
echo $SHELL         # Path to current shell
echo $BASH_VERSION  # Bash version
echo $0             # Name of current script/shell
```

Path Variables

```
# Executable search path
echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/local/sbin:/usr/sbin:/sbin

# Library search path
echo $LD_LIBRARY_PATH

# Manual page search path
echo $MANPATH
```

Locale Variables

```
# Language and locale settings
echo $LANG          # Default locale
echo $LC_ALL        # Override all locale settings
echo $LC_TIME       # Time format locale
echo $LC_NUMERIC    # Number format locale
```

Setting Environment Variables

Temporary Variables (Current Session)

```
# Set variable for current session
$ export MY_VAR="Hello World"
$ echo $MY_VAR
Hello World

# Alternative syntax
$ MY_VAR="Hello World"
$ export MY_VAR
```

Permanent Variables

```
# Add to ~/.bashrc for user-specific variables
echo 'export MY_APP_PATH="/opt/myapp"' >> ~/.bashrc

# Add to /etc/environment for system-wide variables (Ubuntu/Debian)
echo 'MY_GLOBAL_VAR="value"' | sudo tee -a /etc/environment

# Add to /etc/profile for system-wide variables (all systems)
echo 'export GLOBAL_PATH="/usr/local/custom"' | sudo tee -a /etc/profile
```

Unsetting Variables

```
# Remove environment variable
$ unset MY_VAR
$ echo $MY_VAR
# (empty output)

# Remove from environment but keep as shell variable
$ export -n MY_VAR
```

Shell Variables vs Environment Variables

Shell Variables

Only available in the current shell:

```
# Shell variable (not exported)
$ my_shell_var="local value"
$ echo $my_shell_var
local value

# Start subshell - variable not available
$ bash
$ echo $my_shell_var
# (empty)
$ exit
```

Environment Variables

Available to child processes:

```
# Environment variable (exported)
$ export my_env_var="global value"
$ echo $my_env_var
global value

# Start subshell - variable is available
$ bash
$ echo $my_env_var
global value
$ exit
```

Special Variables

Bash provides several special variables with predefined meanings:

Positional Parameters

```
#!/bin/bash
# script_args.sh
echo "Script name: $0"
echo "First argument: $1"
echo "Second argument: $2"
echo "All arguments: $@"
echo "Number of arguments: $#"
echo "All arguments as single string: $*"
```

Process Variables

```
#!/bin/bash
echo "Current process ID: $$"
echo "Parent process ID: $PPID"
echo "Last background process ID: $!"
echo "Exit status of last command: $?"
```

Test Special Variables

```
$ ./script_args.sh hello world test
Script name: ./script_args.sh
First argument: hello
Second argument: world
All arguments: hello world test
Number of arguments: 3
All arguments as single string: hello world test
```

Configuration Files

Bash reads various configuration files at startup, depending on how it's invoked:

Login Shell Files (in order)

1. `/etc/profile` - System-wide login settings
2. `~/.bash_profile` - User login settings
3. `~/.bash_login` - Alternative user login settings
4. `~/.profile` - POSIX-compatible user settings

Non-Login Interactive Shell Files

1. `/etc/bash.bashrc` - System-wide interactive settings
2. `~/.bashrc` - User interactive settings

Non-Interactive Shell Files

- Uses `$BASH_ENV` if set

Logout Files

- `~/.bash_logout` - Executed when login shell exits

Configuration File Examples

~/.bashrc

```
# ~/.bashrc - User interactive shell configuration

# Source global definitions
if [ -f /etc/bashrc ]; then
    . /etc/bashrc
fi

# User specific aliases and functions
alias ll='ls -la'
alias la='ls -A'
alias l='ls -CF'

# Custom prompt
PS1='\u@\h:\w\$ '

# Environment variables
export EDITOR=vim
export PAGER=less

# Add personal bin to PATH
if [ -d "$HOME/bin" ]; then
    PATH="$HOME/bin:$PATH"
fi

# Custom functions
mkcd() {
    mkdir -p "$1" && cd "$1"
}

# History settings
HISTSIZE=1000
HISTFILESIZE=2000
HISTCONTROL=ignoredups
```

~/.bash_profile

```
# ~/.bash_profile - User login shell configuration

# Get the aliases and functions
if [ -f ~/.bashrc ]; then
    . ~/.bashrc
```

```
fi

# User specific environment and startup programs
export PATH="$HOME/.local/bin:$PATH"

# Start SSH agent
if [ -z "$SSH_AUTH_SOCK" ]; then
    eval $(ssh-agent -s)
fi

# Welcome message
echo "Welcome, $USER!"
echo "Today is $(date)"
```

Prompt Customization

The shell prompt is controlled by the `PS1` variable:

Basic Prompt Variables

```
# Default prompt
PS1='\u@\h:\w\$ '
# Results in: user@hostname:/current/path$

# Prompt escape sequences
\u  # Username
\h  # Hostname (short)
\H  # Hostname (full)
\w  # Current working directory
\W  # Basename of current directory
\d  # Date
\t  # Time (24-hour)
\T  # Time (12-hour)
\$  # $ for regular user, # for root
```

Colorized Prompt

```
# Colors
RED='\033[0;31m'
GREEN='\033[0;32m'
YELLOW='\033[1;33m'
BLUE='\033[0;34m'
```

```
NC='\033[0m' # No Color

# Colorized prompt
PS1="${GREEN}\u@\h${NC}:${BLUE}\w${NC}\$ "
```

Advanced Prompt with Git Status

```
# Function to show git branch
git_branch() {
    git branch 2>/dev/null | grep '^*' | colrm 1 2
}

# Prompt with git branch
PS1='\u@\h:\w$(git_branch)\$ '
```

PATH Management

The PATH variable determines where the shell looks for executable files:

Viewing PATH

```
# Display PATH
$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/local/sbin:/usr/sbin:/sbin

# Display PATH in readable format
$ echo $PATH | tr ':' '\n'
/usr/local/bin
/usr/bin
/bin
/usr/local/sbin
/usr/sbin
/sbin
```

Modifying PATH

```
# Add directory to beginning of PATH
export PATH="/new/directory:$PATH"

# Add directory to end of PATH
export PATH="$PATH:/new/directory"
```

```
# Add multiple directories
export PATH="/dir1:/dir2:$PATH:/dir3"
```

PATH Best Practices

```
# Check if directory exists before adding
if [ -d "$HOME/bin" ]; then
    export PATH="$HOME/bin:$PATH"
fi

# Remove duplicate entries
PATH=$(echo "$PATH" | awk -v RS=':' -v ORS=":" '!a[$1]++{if (NR > 1) printf ORS; printf $a[$
```

Shell Options and Settings

Set Options

```
# Display all set options
$ set -o

# Enable options
set -o errexit      # Exit on error
set -o nounset      # Exit on undefined variable
set -o pipefail     # Exit on pipe failure

# Disable options
set +o errexit      # Disable exit on error

# Short form
set -e              # Same as set -o errexit
set -u              # Same as set -o nounset
set -x              # Enable debug mode
```

Shopt Options

```
# Display all shopt options
$ shopt

# Enable options
shopt -s extglob    # Extended globbing
shopt -s globstar   # Recursive globbing (**)
```

```
shopt -s nocaseglob # Case-insensitive globbing

# Disable options
shopt -u extglob    # Disable extended globbing
```

Environment Management Scripts

Environment Setup Script

```
#!/bin/bash
# setup_env.sh - Development environment setup

echo "Setting up development environment..."

# Java environment
export JAVA_HOME="/usr/lib/jvm/java-11-openjdk"
export PATH="$JAVA_HOME/bin:$PATH"

# Node.js environment
export NODE_PATH="/usr/local/lib/node_modules"
export PATH="/usr/local/node/bin:$PATH"

# Python environment
export PYTHONPATH="/usr/local/lib/python3.9/site-packages:$PYTHONPATH"

# Custom application paths
export APP_HOME="/opt/myapp"
export PATH="$APP_HOME/bin:$PATH"

# Development tools
export EDITOR=vim
export BROWSER=firefox

echo "Environment setup complete!"
echo "Java version: $(java -version 2>&1 | head -1)"
echo "Node version: $(node --version 2>/dev/null || echo 'Not installed')"
echo "Python version: $(python3 --version)"
```

Environment Backup and Restore

```
#!/bin/bash
# env_backup.sh - Backup current environment

BACKUP_FILE="env_backup_$(date +%Y%m%d_%H%M%S).sh"

echo "#!/bin/bash" > "$BACKUP_FILE"
echo "# Environment backup created on $(date)" >> "$BACKUP_FILE"
echo "" >> "$BACKUP_FILE"

# Backup important variables
for var in PATH HOME USER SHELL LANG; do
    echo "export $var='${!var}'" >> "$BACKUP_FILE"
done

echo "Environment backed up to $BACKUP_FILE"
```

Debugging Environment Issues

Common Environment Problems

```
# Check if command is found
$ which python3
/usr/bin/python3

$ type python3
python3 is /usr/bin/python3

# Check if variable is set
$ [ -z "$JAVA_HOME" ] && echo "JAVA_HOME not set"

# Debug PATH issues
$ echo $PATH | grep -o '[^:]*' | while read dir; do
    [ -d "$dir" ] || echo "Directory not found: $dir"
done
```

Environment Diagnostic Script

```
#!/bin/bash
# env_diagnostic.sh - Diagnose environment issues

echo "=== Environment Diagnostic ==="
```

```
echo "Date: $(date)"
echo "User: $USER"
echo "Shell: $SHELL"
echo "Bash Version: $BASH_VERSION"
echo

echo "=== PATH Analysis ==="
echo "PATH has $(echo $PATH | tr ':' '\n' | wc -l) directories"
echo $PATH | tr ':' '\n' | while read dir; do
    if [ -d "$dir" ]; then
        echo "  $dir"
    else
        echo "  $dir (not found)"
    fi
done
echo

echo "=== Important Variables ==="
for var in HOME USER SHELL LANG EDITOR PAGER; do
    echo "$var: ${!var:-'(not set)'}"
done
```

Understanding and properly managing the Bash environment is essential for creating robust scripts and maintaining a productive development environment. The environment affects everything from command execution to script behavior, making it a fundamental aspect of shell scripting.

Bash Features

Bash offers a rich set of features that make it powerful for both interactive use and scripting. Understanding these features will help you write more efficient and effective scripts.

Interactive Features

Command History

Bash maintains a history of commands you've executed:

```
# View command history
$ history
1001  ls -la
1002  cd /home
1003  pwd
1004  echo "hello world"
1005  history

# Execute previous command
$ !!

# Execute command by number
$ !1003

# Search history
$ !pw
pwd

# Reverse search (Ctrl+R)
(reverse-i-search)`ls': ls -la
```

History Configuration

```
# In ~/.bashrc
HISTSIZE=1000          # Commands in memory
HISTFILESIZE=2000      # Commands in history file
HISTCONTROL=ignoredups # Ignore duplicate commands
```

```
# Don't save certain commands
HISTIGNORE="ls:pwd:exit:clear"
```

Tab Completion

Bash provides intelligent tab completion:

```
# Complete commands
$ ec<TAB>
echo

# Complete file names
$ ls /usr/b<TAB>
bin/

# Complete variable names
$ echo $HO<TAB>
$HOME

# Complete options
$ ls --<TAB><TAB>
--all                     --escape
--almost-all              --file-type
--author                  --format
```

Custom Completion

```
# Add to ~/.bashrc
complete -W "start stop restart status" service
complete -d cd  # Only directories for cd
```

Command Line Editing

Bash uses readline library for command editing:

```
# Cursor movement
Ctrl+A    # Beginning of line
Ctrl+E    # End of line
Ctrl+B    # Back one character
Ctrl+F    # Forward one character
Alt+B     # Back one word
Alt+F     # Forward one word
```

```
# Editing
Ctrl+D    # Delete character under cursor
Ctrl+H    # Delete character before cursor
Ctrl+K    # Kill from cursor to end of line
Ctrl+U    # Kill from cursor to beginning of line
Ctrl+W    # Kill word before cursor
Alt+D     # Kill word after cursor

# History navigation
Ctrl+P    # Previous command
Ctrl+N    # Next command
Ctrl+R    # Reverse search
```

Job Control

Bash provides sophisticated job control capabilities:

Background Jobs

```
# Run command in background
$ long_running_command &
[1] 12345

# List active jobs
$ jobs
[1]+  Running    long_running_command &

# Bring job to foreground
$ fg %1

# Send job to background
$ bg %1
```

Job Management

```
#!/bin/bash
# job_control_demo.sh

echo "Starting background jobs..."

# Start multiple background jobs
sleep 30 &
```

```
job1=$!

sleep 40 &
job2=$!

echo "Job 1 PID: $job1"
echo "Job 2 PID: $job2"

# Wait for specific job
wait $job1
echo "Job 1 completed"

# Wait for all background jobs
wait
echo "All jobs completed"
```

Process Control

```
# Suspend current process
Ctrl+Z

# Kill current process
Ctrl+C

# Send signals to processes
$ kill -TERM 12345
$ kill -KILL 12345
$ killall process_name
```

Aliases and Functions

Aliases

Simple command shortcuts:

```
# Define aliases
alias ll='ls -la'
alias la='ls -A'
alias l='ls -CF'
alias grep='grep --color=auto'

# Use aliases
$ ll
total 24
```

```
drwxr-xr-x  3 user user 4096 Jan 15 10:30 .
drwxr-xr-x 25 user user 4096 Jan 15 10:25 ..

# List all aliases
$ alias

# Remove alias
$ unalias ll
```

Functions

More complex reusable code:

```
# Simple function
greet() {
    echo "Hello, $1!"
}

# Function with multiple parameters
backup_file() {
    local file="$1"
    local backup_dir="$2"

    if [ -f "$file" ]; then
        cp "$file" "$backup_dir/$(basename "$file").bak"
        echo "Backed up $file"
    else
        echo "File $file not found"
        return 1
    fi
}

# Use functions
$ greet "World"
Hello, World!

$ backup_file "/etc/hosts" "/backup"
Backed up /etc/hosts
```

Variable Features

Variable Types

```
# Regular variables
name="John"
age=25

# Arrays
fruits=("apple" "banana" "orange")
echo ${fruits[0]}  # apple
echo ${fruits[@]}  # all elements

# Associative arrays (Bash 4+)
declare -A colors
colors[red]="#FF0000"
colors[green]="#00FF00"
echo ${colors[red]}  # #FF0000
```

Parameter Expansion

```
filename="document.txt"

# Basic expansion
echo $filename          # document.txt
echo ${filename}        # document.txt

# Length
echo ${#filename}       # 12

# Substring
echo ${filename:0:8}    # document

# Remove from end
echo ${filename%.*}     # document

# Remove from beginning
echo ${filename##*.}    # txt

# Replace
echo ${filename/txt/pdf}  # document.pdf
```

Default Values

```
# Use default if variable is unset
echo ${name:-"Anonymous"}

# Set default if variable is unset
echo ${name:="Default Name"}

# Error if variable is unset
echo ${name:?"Variable name is required"}

# Use alternative if variable is set
echo ${name:+"Name is set"}
```

Pattern Matching and Globbing

Basic Wildcards

```
# Match any characters
$ ls *.txt
file1.txt  file2.txt  document.txt

# Match single character
$ ls file?.txt
file1.txt  file2.txt

# Match character ranges
$ ls file[1-3].txt
file1.txt  file2.txt  file3.txt

# Match character sets
$ ls file[abc].txt
filea.txt  fileb.txt  filec.txt
```

Extended Globbing

```
# Enable extended globbing
shopt -s extglob

# Match one of several patterns
$ ls *.@(txt|pdf|doc)

# Match zero or one occurrence
```

```
$ ls file?(s).txt

# Match zero or more occurrences
$ ls file*(s).txt

# Match one or more occurrences
$ ls file+(s).txt

# Match anything except pattern
$ ls !(*.tmp)
```

Globstar (Bash 4+)

```
# Enable globstar
shopt -s globstar

# Recursive matching
$ ls **/*.txt
dir1/file1.txt
dir1/subdir/file2.txt
dir2/file3.txt
```

Brace Expansion

Sequence Generation

```
# Number sequences
$ echo {1..10}
1 2 3 4 5 6 7 8 9 10

$ echo {01..10}
01 02 03 04 05 06 07 08 09 10

# Letter sequences
$ echo {a..z}
a b c d e f g h i j k l m n o p q r s t u v w x y z

# Step sequences (Bash 4+)
$ echo {1..10..2}
1 3 5 7 9
```

String Expansion

```
# Multiple strings
$ echo {cat,dog,bird}
cat dog bird

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

# File operations
$ cp file.txt{,.bak}  # Same as: cp file.txt file.txt.bak
$ mkdir -p project/{src,docs,tests}
```

Process Substitution

Input Substitution

```
# Compare output of two commands
$ diff <(ls dir1) <(ls dir2)

# Use command output as input
$ while read line; do
    echo "Processing: $line"
done < <(find /var/log -name "*.log")
```

Output Substitution

```
# Send output to multiple commands
$ echo "data" | tee >(command1) >(command2)

# Log to file and display
$ make 2>&1 | tee >(logger -t build)
```

Command Substitution

Modern Syntax

```
# Preferred method
current_date=$(date)
file_count=$(ls | wc -l)
user_home=$(eval echo ~$USER)

echo "Today is $current_date"
echo "Files in directory: $file_count"
```

Legacy Syntax

```
# Older method (still works)
current_date=`date`
file_count=`ls | wc -l`
```

Nested Substitution

```
# Complex nested commands
backup_name="backup_$(date +%Y%m%d)_$(hostname).tar.gz"
echo $backup_name  # backup_20240115_myserver.tar.gz
```

Arithmetic Operations

Arithmetic Expansion

```
# Basic arithmetic
result=$((5 + 3))
echo $result  # 8

# Variables in arithmetic
a=10
b=5
echo $((a + b))   # 15
echo $((a * b))   # 50
echo $((a / b))   # 2
echo $((a % b))   # 0
```

Arithmetic Commands

```
# let command
let "result = 5 + 3"
echo $result  # 8

# expr command (external)
result=$(expr 5 + 3)
echo $result  # 8
```

Advanced Arithmetic

```
# Increment/decrement
counter=0
((counter++))
echo $counter  # 1

((counter += 5))
echo $counter  # 6

# Comparison in arithmetic
if ((counter > 5)); then
    echo "Counter is greater than 5"
fi
```

Conditional Expressions

Test Command

```
# File tests
if [ -f "file.txt" ]; then
    echo "File exists"
fi

# String tests
if [ "$name" = "John" ]; then
    echo "Hello John"
fi

# Numeric tests
if [ $age -gt 18 ]; then
    echo "Adult"
fi
```

Extended Test

```
# Bash extended test
if [[ $name == "John" ]]; then
    echo "Hello John"
fi

# Pattern matching
if [[ $filename == *.txt ]]; then
    echo "Text file"
fi

# Regular expressions
if [[ $email =~ ^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$ ]]; then
    echo "Valid email"
fi
```

I/O Redirection

Basic Redirection

```
# Redirect stdout
$ echo "Hello" > file.txt

# Redirect stderr
$ command 2> error.log

# Redirect both
$ command > output.log 2>&1
$ command &> output.log  # Bash shortcut
```

Advanced Redirection

```
# Append to file
$ echo "More data" >> file.txt

# Here documents
$ cat << EOF > config.txt
Setting1=value1
Setting2=value2
EOF

# Here strings
```

```
$ grep "pattern" <<< "$variable"
```

Shell Options

Set Options

```
# Exit on error
set -e

# Exit on undefined variable
set -u

# Exit on pipe failure
set -o pipefail

# Combine options
set -euo pipefail
```

Shopt Options

```
# Enable extended globbing
shopt -s extglob

# Enable case-insensitive globbing
shopt -s nocaseglob

# Enable recursive globbing
shopt -s globstar

# List all options
shopt
```

These features make Bash a powerful and flexible shell for both interactive use and scripting. Understanding and utilizing these features will significantly improve your productivity and the quality of your scripts.

What is Bash?

Bash (Bourne Again Shell) is the most widely used shell in Linux and Unix systems. It's an enhanced version of the original Bourne shell (sh) with many additional features that make it powerful for both interactive use and scripting.

History and Background

The Evolution

1. **Bourne Shell (sh)** - 1977: Original Unix shell by Stephen Bourne
2. **C Shell (csh)** - 1978: Added command history and job control
3. **Korn Shell (ksh)** - 1983: Combined features of sh and csh
4. **Bash** - 1989: GNU's "Bourne Again Shell" by Brian Fox

Why "Bourne Again"?

- **Pun**: Play on "born again" (reborn)
- **Compatibility**: Maintains compatibility with original Bourne shell
- **Enhancement**: Adds modern features while preserving classic functionality

Bash Characteristics

POSIX Compliance

Bash is largely POSIX-compliant, meaning scripts written for POSIX shells should work in Bash:

```
#!/bin/sh
# This POSIX script works in bash
echo "Hello, World!"
if [ -f "/etc/passwd" ]; then
    echo "Password file exists"
fi
```

GNU Project

- **Free Software**: Part of the GNU operating system
- **Open Source**: Source code freely available
- **Community Driven**: Developed and maintained by volunteers

Bash Versions

Check Your Bash Version

```
$ bash --version
GNU bash, version 5.1.16(1)-release (x86_64-pc-linux-gnu)

# Or from within bash
$ echo $BASH_VERSION
5.1.16(1)-release

# Short version
$ echo ${BASH_VERSION%%.*}
5
```

Major Version Differences

Bash 3.x (2004-2009)

- Regular expressions in conditional expressions
- Process substitution improvements

Bash 4.x (2009-2020)

- Associative arrays
- Case modification operators
- Globstar (**) pattern matching

Bash 5.x (2019-present)

- Improved performance
- New variable expansion features
- Enhanced debugging capabilities

Version-Specific Features

```
#!/bin/bash

# Check bash version for feature compatibility
if [ "${BASH_VERSION%%.*}" -ge 4 ]; then
    # Bash 4+ features
    declare -A assoc_array
    assoc_array[key]="value"
    echo "Using associative arrays"
else
    # Fallback for older bash
    echo "Using indexed arrays"
fi
```

Bash vs Other Shells

Bash vs Bourne Shell (sh)

```
# Bash-specific features not in sh
echo ${BASH_VERSION}        # Bash version variable
echo ${!var*}               # Variable name expansion
declare -A array            # Associative arrays
[[ $var =~ regex ]]         # Regular expression matching
```

Bash vs Zsh

```
# Bash
$ echo $SHELL
/bin/bash

# Zsh has more advanced features
$ echo $SHELL
/usr/bin/zsh
```

Bash vs Fish

```
# Bash syntax
if [ $status -eq 0 ]; then
    echo "Success"
fi
```

```
# Fish syntax (different)
if test $status -eq 0
    echo "Success"
end
```

Where Bash is Used

Default Shell

Bash is the default shell on most Linux distributions:

```
# Check default shell
$ echo $SHELL
/bin/bash

# Check available shells
$ cat /etc/shells
/bin/sh
/bin/bash
/usr/bin/bash
/bin/rbash
/usr/bin/rbash
/bin/dash
/usr/bin/dash
```

System Scripts

Many system scripts use Bash:

```
# System initialization scripts
$ head -1 /etc/init.d/ssh
#!/bin/bash

# Package management scripts
$ head -1 /usr/bin/apt-get
#!/bin/bash
```

Development and DevOps

```
# Build scripts
#!/bin/bash
make clean
make all
```

```
make install

# Deployment scripts
#!/bin/bash
git pull origin main
npm install
npm run build
systemctl restart myapp
```

Bash Capabilities

Interactive Features

```
# Command history
$ history | tail -5
1001  ls -la
1002  cd /home
1003  pwd
1004  echo "hello"
1005  history | tail -5

# Tab completion
$ ls /usr/b<TAB>
bin/

# Command editing
# Use arrow keys, Ctrl+A (beginning), Ctrl+E (end)
```

Programming Features

```
#!/bin/bash

# Variables
name="John"
age=25

# Arrays
fruits=("apple" "banana" "orange")

# Functions
greet() {
    echo "Hello, $1!"
}
```

```
# Control structures
if [ $age -gt 18 ]; then
    echo "Adult"
fi

# Loops
for fruit in "${fruits[@]}"; do
    echo "Fruit: $fruit"
done
```

Advanced Features

```
#!/bin/bash

# Process substitution
diff <(ls dir1) <(ls dir2)

# Parameter expansion
filename="document.txt"
echo ${filename%.*}     # document
echo ${filename##*.}    # txt

# Regular expressions
if [[ $email =~ ^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$ ]]; then
    echo "Valid email"
fi
```

Bash Configuration

Configuration Files

```
# System-wide configuration
/etc/bash.bashrc        # All users
/etc/profile           # Login shells

# User-specific configuration
~/.bashrc              # Interactive non-login shells
~/.bash_profile        # Login shells
~/.bash_logout         # Logout cleanup
```

Customization Example

```
# ~/.bashrc
# Custom prompt
PS1='\u@\h:\w\$ '

# Aliases
alias ll='ls -la'
alias grep='grep --color=auto'

# Functions
mkcd() {
    mkdir -p "$1" && cd "$1"
}

# Environment variables
export EDITOR=vim
export PATH="$HOME/bin:$PATH"
```

Bash Built-in Commands

Bash includes many built-in commands for efficiency:

```
# File operations
$ type cd
cd is a shell builtin

$ type echo
echo is a shell builtin

# List all built-ins
$ help
# or
$ compgen -b
```

Common Built-ins

```
# Navigation
cd /path/to/directory
pwd

# Variables
export VAR=value
unset VAR
```

```
# Control
exit 0
return 1

# Information
type command
which command
help command
```

Bash Scripting Advantages

1. Ubiquity

- Available on virtually all Unix-like systems
- Default shell on most Linux distributions
- Consistent behavior across platforms

2. Integration

- Seamless integration with system commands
- Easy to call external programs
- Natural for system administration tasks

3. Learning Curve

- Familiar to anyone who uses the command line
- Gradual learning from simple commands to complex scripts
- Extensive documentation and community support

4. Power and Flexibility

```
#!/bin/bash
# Powerful one-liners
find /var/log -name "*.log" -mtime +7 -exec rm {} \;

# Complex data processing
awk '/ERROR/ {count++} END {print "Errors:", count}' logfile.txt

# System monitoring
ps aux | awk '$3 > 80 {print $2, $11}' | head -10
```

When to Use Bash

Perfect For:

- **System administration**: User management, backups, monitoring
- **DevOps tasks**: Deployment, CI/CD pipelines
- **File processing**: Batch operations, log analysis
- **Glue scripts**: Connecting different tools and programs
- **Quick automation**: Tasks that don't require complex logic

Consider Alternatives For:

- **Complex applications**: Use Python, Go, or other languages
- **Performance-critical tasks**: Compiled languages are faster
- **Cross-platform GUI apps**: Use appropriate frameworks
- **Heavy mathematical computations**: Use specialized tools

Getting Started with Bash

Check if Bash is Available

```
$ which bash
/bin/bash

$ bash --version
GNU bash, version 5.1.16(1)-release
```

Start Using Bash

```
# If not your default shell
$ bash

# Make bash your default shell
$ chsh -s $(which bash)
```

First Bash Script

```
#!/bin/bash
echo "Welcome to Bash scripting!"
echo "Bash version: $BASH_VERSION"
echo "Today is: $(date)"
```

Bash is more than just a command interpreter—it's a powerful programming environment that bridges the gap between simple command-line operations and full-featured programming languages. Its widespread adoption, extensive features, and excellent documentation make it an essential tool for anyone working with Linux or Unix systems.

Variables

Variable Basics

Variables are fundamental building blocks in Bash scripting. They allow you to store data, make scripts flexible, and create reusable code. Understanding how to work with variables effectively is essential for writing robust shell scripts.

What are Variables?

Variables are named storage locations that hold data. In Bash, variables can store: - Text strings - Numbers (treated as strings) - Command output - File paths - Configuration values

Variable Declaration and Assignment

Basic Syntax

```
# Variable assignment (no spaces around =)
variable_name="value"

# Examples
name="John Doe"
age=25
city="New York"
is_active=true
```

Important Rules

1. **No spaces** around the equals sign
2. **Case sensitive**: `name` and `Name` are different variables
3. **No type declaration** needed
4. **Quote values** with spaces or special characters

Common Mistakes

```
# Wrong - spaces around =
name = "John"        # Error!

# Wrong - unquoted value with spaces
name=John Doe        # Error!

# Correct
name="John Doe"      # Correct
```

Variable Naming Conventions

Valid Variable Names

```
# Good variable names
username="john"
user_name="john"
userName="john"
USER_NAME="john"
file1="document.txt"
_private_var="secret"
```

Invalid Variable Names

```
# Invalid - cannot start with number
1name="john"         # Error!

# Invalid - contains special characters
user-name="john"     # Error!
user@name="john"     # Error!

# Invalid - reserved words
for="value"          # Error!
if="value"           # Error!
```

Best Practices

```
# Use descriptive names
config_file="/etc/myapp.conf"
max_connections=100
database_host="localhost"
```

```
# Use uppercase for constants
readonly PI=3.14159
readonly MAX_RETRIES=3

# Use lowercase for regular variables
current_user="john"
temp_directory="/tmp"
```

Accessing Variable Values

Basic Access

```
name="Alice"

# Access variable value
echo $name           # Alice
echo ${name}         # Alice (preferred for clarity)
```

When to Use Braces

```
name="Alice"

# Without braces - can be ambiguous
echo $names          # Empty (variable 'names' doesn't exist)

# With braces - clear separation
echo ${name}s        # Alices

# Concatenation examples
prefix="file"
echo ${prefix}_backup.txt    # file_backup.txt
echo $prefix_backup.txt      # Empty (looks for variable 'prefix_backup')
```

Variable Types and Behavior

String Variables

```
# Simple strings
greeting="Hello"
message="Welcome to Bash scripting"

# Strings with special characters
path="/home/user/My Documents"
command="ls -la"

# Empty strings
empty_var=""
another_empty=""
```

Numeric Variables

```
# Numbers are stored as strings but can be used in arithmetic
count=10
price=19.99
year=2024

# Arithmetic operations
total=$((count * 2))
echo $total          # 20

# Comparison
if [ $count -gt 5 ]; then
    echo "Count is greater than 5"
fi
```

Boolean-like Variables

```
# Bash doesn't have true boolean types
# Use strings to represent boolean values
is_enabled="true"
debug_mode="false"
has_permission="yes"

# Check boolean-like values
if [ "$is_enabled" = "true" ]; then
    echo "Feature is enabled"
fi
```

Variable Scope

Global Variables

```
#!/bin/bash
# Global variable - accessible everywhere
global_var="I'm global"

function test_function() {
    echo "Inside function: $global_var"
}

echo "Outside function: $global_var"
test_function
```

Local Variables

```
#!/bin/bash
function test_local() {
    local local_var="I'm local"
    global_var="Modified global"

    echo "Local variable: $local_var"
    echo "Global variable: $global_var"
}

global_var="Original global"
echo "Before function: $global_var"

test_local

echo "After function: $global_var"
echo "Local variable outside: $local_var"  # Empty
```

Reading User Input

Basic Input

```
#!/bin/bash
echo "What's your name?"
read name
echo "Hello, $name!"
```

Input with Prompt

```
#!/bin/bash
read -p "Enter your age: " age
echo "You are $age years old"
```

Multiple Variables

```
#!/bin/bash
echo "Enter first and last name:"
read first_name last_name
echo "Hello, $first_name $last_name!"
```

Reading into Array

```
#!/bin/bash
echo "Enter three colors:"
read -a colors
echo "First color: ${colors[0]}"
echo "All colors: ${colors[@]}"
```

Silent Input (Passwords)

```
#!/bin/bash
read -s -p "Enter password: " password
echo  # New line after hidden input
echo "Password length: ${#password}"
```

Variable Assignment Methods

Direct Assignment

```
name="John"
age=25
```

Command Substitution

```
# Modern syntax (preferred)
current_date=$(date)
file_count=$(ls | wc -l)
current_user=$(whoami)

# Legacy syntax (still works)
current_date=`date`
file_count=`ls | wc -l`
```

Reading from Files

```
# Read first line of file
first_line=$(head -n1 /etc/passwd)

# Read entire file content
file_content=$(cat /etc/hostname)

# Read with error handling
if config=$(cat config.txt 2>/dev/null); then
    echo "Config loaded: $config"
else
    echo "Failed to read config file"
fi
```

Arithmetic Assignment

```
# Simple arithmetic
count=5
count=$((count + 1))
echo $count          # 6

# Using let command
let count=count+1
echo $count          # 7

# Increment/decrement
((count++))
echo $count          # 8
```

Variable Manipulation

String Length

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

Substring Extraction

```
text="Hello World"

# Extract substring: ${variable:start:length}
echo ${text:0:5}     # Hello
echo ${text:6}       # World
echo ${text:6:3}     # Wor
echo ${text: -5}     # World (note the space before -)
```

Case Conversion

```
text="Hello World"

# Convert to uppercase (Bash 4+)
echo ${text^^}       # HELLO WORLD

# Convert to lowercase (Bash 4+)
echo ${text,,}       # hello world

# Convert first character
echo ${text^}        # Hello World
echo ${text,}        # hello World
```

Pattern Replacement

```
filename="document.txt"

# Replace first occurrence
echo ${filename/txt/pdf}     # document.pdf

# Replace all occurrences
echo ${filename//t/T}        # documenT.TxT

# Remove from beginning
```

```
echo ${filename#doc}          # ument.txt

# Remove from end
echo ${filename%.txt}         # document
```

Default Values and Error Handling

Default Values

```
# Use default if variable is unset or empty
name=${name:-"Anonymous"}
echo "Hello, $name"

# Set default if variable is unset
name=${name:="Default Name"}

# Use alternative if variable is set
message=${name:+"Hello, $name"}
```

Error on Unset Variables

```
# Exit with error if variable is unset
required_var=${required_var:?"Variable required_var must be set"}

# Custom error message
database_host=${database_host:?"Error: DATABASE_HOST environment variable is required"}
```

Practical Example

```
#!/bin/bash
# Configuration script with defaults

# Set defaults
config_file=${CONFIG_FILE:-"/etc/myapp.conf"}
log_level=${LOG_LEVEL:-"INFO"}
max_connections=${MAX_CONNECTIONS:-"100"}

echo "Configuration:"
echo "Config file: $config_file"
echo "Log level: $log_level"
echo "Max connections: $max_connections"
```

```
# Validate required variables
database_url=${DATABASE_URL:?"Error: DATABASE_URL must be set"}
api_key=${API_KEY:?"Error: API_KEY must be set"}
```

Variable Arrays (Preview)

Simple Array

```
# Create array
fruits=("apple" "banana" "orange")

# Access elements
echo ${fruits[0]}    # apple
echo ${fruits[1]}    # banana

# All elements
echo ${fruits[@]}    # apple banana orange

# Array length
echo ${#fruits[@]}   # 3
```

Best Practices

1. Always Quote Variables

```
# Good - prevents word splitting
if [ "$name" = "John Doe" ]; then
    echo "Hello, $name"
fi

# Bad - can break with spaces
if [ $name = "John Doe" ]; then
    echo "Hello, $name"
fi
```

2. Use Meaningful Names

```
# Good
database_connection_string="mysql://localhost:3306/mydb"
max_retry_attempts=3
```

```
# Bad
dcs="mysql://localhost:3306/mydb"
mra=3
```

3. Initialize Variables

```
# Good - initialize variables
count=0
total=0
error_message=""

# Process data
for file in *.txt; do
    ((count++))
    # ... processing ...
done
```

4. Use readonly for Constants

```
# Constants should be readonly
readonly PI=3.14159
readonly MAX_USERS=1000
readonly CONFIG_DIR="/etc/myapp"

# This will cause an error
# PI=3.14  # Error: PI is readonly
```

5. Validate Input

```
#!/bin/bash
read -p "Enter a number: " number

# Validate input
if [[ ! $number =~ ^[0-9]+$ ]]; then
    echo "Error: Please enter a valid number"
    exit 1
fi

echo "You entered: $number"
```

Common Variable Patterns

Configuration Variables

```
#!/bin/bash
# Application configuration

# Default configuration
APP_NAME="MyApplication"
APP_VERSION="1.0.0"
CONFIG_FILE="/etc/${APP_NAME,,}.conf"
LOG_FILE="/var/log/${APP_NAME,,}.log"
PID_FILE="/var/run/${APP_NAME,,}.pid"

# Load configuration file if it exists
if [ -f "$CONFIG_FILE" ]; then
    source "$CONFIG_FILE"
fi
```

Temporary Variables

```
#!/bin/bash
# Temporary file handling

temp_dir=$(mktemp -d)
temp_file=$(mktemp)

# Cleanup function
cleanup() {
    rm -rf "$temp_dir"
    rm -f "$temp_file"
}

# Set trap to cleanup on exit
trap cleanup EXIT

# Use temporary files
echo "Processing data..." > "$temp_file"
```

Understanding variable basics is crucial for Bash scripting. Variables make your scripts flexible, maintainable, and powerful. Practice these concepts and patterns to build a solid foundation for more advanced scripting techniques.

Control

Conditional Statements

Conditional statements allow your scripts to make decisions and execute different code paths based on conditions. They are essential for creating intelligent, responsive scripts that can handle various scenarios.

The if Statement

The `if` statement is the most basic conditional construct in Bash.

Basic Syntax

```
if [ condition ]; then
    # commands to execute if condition is true
fi
```

Simple Example

```
#!/bin/bash
age=20

if [ $age -ge 18 ]; then
    echo "You are an adult"
fi
```

if-else Statement

Add an alternative path when the condition is false:

```
#!/bin/bash
age=16

if [ $age -ge 18 ]; then
    echo "You are an adult"
else
    echo "You are a minor"
```

```
fi
```

if-elif-else Statement

Handle multiple conditions:

```
#!/bin/bash
score=85

if [ $score -ge 90 ]; then
    echo "Grade: A"
elif [ $score -ge 80 ]; then
    echo "Grade: B"
elif [ $score -ge 70 ]; then
    echo "Grade: C"
elif [ $score -ge 60 ]; then
    echo "Grade: D"
else
    echo "Grade: F"
fi
```

Test Conditions

Numeric Comparisons

```
#!/bin/bash
num1=10
num2=20

# Equal
if [ $num1 -eq $num2 ]; then
    echo "Numbers are equal"
fi

# Not equal
if [ $num1 -ne $num2 ]; then
    echo "Numbers are not equal"
fi

# Greater than
if [ $num1 -gt $num2 ]; then
    echo "$num1 is greater than $num2"
fi
```

```
# Less than
if [ $num1 -lt $num2 ]; then
    echo "$num1 is less than $num2"
fi

# Greater than or equal
if [ $num1 -ge $num2 ]; then
    echo "$num1 is greater than or equal to $num2"
fi

# Less than or equal
if [ $num1 -le $num2 ]; then
    echo "$num1 is less than or equal to $num2"
fi
```

String Comparisons

```
#!/bin/bash
name1="Alice"
name2="Bob"
empty_string=""

# String equality
if [ "$name1" = "$name2" ]; then
    echo "Names are the same"
else
    echo "Names are different"
fi

# String inequality
if [ "$name1" != "$name2" ]; then
    echo "Names are different"
fi

# String length (non-zero)
if [ -n "$name1" ]; then
    echo "Name1 is not empty"
fi

# String length (zero)
if [ -z "$empty_string" ]; then
    echo "String is empty"
fi

# Lexicographic comparison
```

```
if [[ "$name1" < "$name2" ]]; then
    echo "$name1 comes before $name2 alphabetically"
fi
```

File Tests

```
#!/bin/bash
filename="test.txt"
directory="mydir"

# File exists
if [ -e "$filename" ]; then
    echo "File exists"
fi

# Regular file
if [ -f "$filename" ]; then
    echo "It's a regular file"
fi

# Directory
if [ -d "$directory" ]; then
    echo "It's a directory"
fi

# Readable
if [ -r "$filename" ]; then
    echo "File is readable"
fi

# Writable
if [ -w "$filename" ]; then
    echo "File is writable"
fi

# Executable
if [ -x "$filename" ]; then
    echo "File is executable"
fi

# File size greater than zero
if [ -s "$filename" ]; then
    echo "File is not empty"
fi
```

```
# Symbolic link
if [ -L "$filename" ]; then
    echo "It's a symbolic link"
fi
```

Advanced Test Constructs

Double Brackets [[ ]]

The `[[ ]]` construct provides more features than single brackets:

```
#!/bin/bash
name="Alice"
number="123"

# Pattern matching
if [[ $name == A* ]]; then
    echo "Name starts with A"
fi

# Regular expressions
if [[ $number =~ ^[0-9]+$ ]]; then
    echo "It's a number"
fi

# Multiple conditions with && and ||
if [[ $name == "Alice" && $number -gt 100 ]]; then
    echo "Name is Alice and number is greater than 100"
fi

if [[ $name == "Alice" || $name == "Bob" ]]; then
    echo "Name is either Alice or Bob"
fi
```

Arithmetic Evaluation (( ))

For numeric comparisons, you can use arithmetic evaluation:

```
#!/bin/bash
x=10
y=20

if (( x < y )); then
    echo "$x is less than $y"
```

```
fi

if (( x + y == 30 )); then
    echo "Sum is 30"
fi

if (( x % 2 == 0 )); then
    echo "$x is even"
fi
```

Logical Operators

AND Operator (&&)

```
#!/bin/bash
age=25
has_license=true

if [ $age -ge 18 ] && [ "$has_license" = "true" ]; then
    echo "Can drive"
fi

# Alternative syntax
if [[ $age -ge 18 && $has_license == "true" ]]; then
    echo "Can drive"
fi
```

OR Operator (||)

```
#!/bin/bash
day="Saturday"

if [ "$day" = "Saturday" ] || [ "$day" = "Sunday" ]; then
    echo "It's weekend!"
fi

# Alternative syntax
if [[ $day == "Saturday" || $day == "Sunday" ]]; then
    echo "It's weekend!"
fi
```

NOT Operator (!)

```
#!/bin/bash
filename="test.txt"

# Negate condition
if [ ! -f "$filename" ]; then
    echo "File does not exist"
fi

# Alternative
if ! [ -f "$filename" ]; then
    echo "File does not exist"
fi
```

Practical Examples

File Backup Script

```
#!/bin/bash
source_file="$1"
backup_dir="/backup"

# Check if source file is provided
if [ $# -eq 0 ]; then
    echo "Usage: $0 <source_file>"
    exit 1
fi

# Check if source file exists
if [ ! -f "$source_file" ]; then
    echo "Error: Source file '$source_file' does not exist"
    exit 1
fi

# Check if backup directory exists
if [ ! -d "$backup_dir" ]; then
    echo "Creating backup directory..."
    mkdir -p "$backup_dir"
fi

# Create backup
backup_name="$(basename "$source_file").backup.$(date +%Y%m%d_%H%M%S)"
if cp "$source_file" "$backup_dir/$backup_name"; then
```

```
    echo "Backup created: $backup_dir/$backup_name"
else
    echo "Error: Failed to create backup"
    exit 1
fi
```

System Health Check

```
#!/bin/bash
# System health check script

echo "=== System Health Check ==="

# Check disk usage
disk_usage=$(df / | awk 'NR==2 {print $5}' | sed 's/%//')
if [ $disk_usage -gt 90 ]; then
    echo "WARNING: Disk usage is ${disk_usage}%"
elif [ $disk_usage -gt 80 ]; then
    echo "CAUTION: Disk usage is ${disk_usage}%"
else
    echo "OK: Disk usage is ${disk_usage}%"
fi

# Check memory usage
memory_usage=$(free | awk 'NR==2{printf "%.0f", $3*100/$2}')
if [ $memory_usage -gt 90 ]; then
    echo "WARNING: Memory usage is ${memory_usage}%"
else
    echo "OK: Memory usage is ${memory_usage}%"
fi

# Check if critical services are running
services=("ssh" "nginx" "mysql")
for service in "${services[@]}"; do
    if systemctl is-active --quiet "$service"; then
        echo "OK: $service is running"
    else
        echo "ERROR: $service is not running"
    fi
done
```

User Input Validation

```
#!/bin/bash
# User registration script with validation

echo "User Registration"
echo "=================="

# Get username
while true; do
    read -p "Enter username (3-20 characters, alphanumeric only): " username

    if [[ -z "$username" ]]; then
        echo "Username cannot be empty"
    elif [[ ${#username} -lt 3 ]]; then
        echo "Username must be at least 3 characters"
    elif [[ ${#username} -gt 20 ]]; then
        echo "Username must be no more than 20 characters"
    elif [[ ! $username =~ ^[a-zA-Z0-9]+$ ]]; then
        echo "Username can only contain letters and numbers"
    else
        break
    fi
done

# Get email
while true; do
    read -p "Enter email address: " email

    if [[ -z "$email" ]]; then
        echo "Email cannot be empty"
    elif [[ ! $email =~ ^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$ ]]; then
        echo "Please enter a valid email address"
    else
        break
    fi
done

# Get age
while true; do
    read -p "Enter age (18-120): " age

    if [[ ! $age =~ ^[0-9]+$ ]]; then
        echo "Age must be a number"
    elif [ $age -lt 18 ]; then
        echo "You must be at least 18 years old"
```

```
    elif [ $age -gt 120 ]; then
        echo "Please enter a realistic age"
    else
        break
    fi
done

echo
echo "Registration successful!"
echo "Username: $username"
echo "Email: $email"
echo "Age: $age"
```

Nested Conditionals

You can nest if statements for complex logic:

```
#!/bin/bash
weather="sunny"
temperature=75

if [ "$weather" = "sunny" ]; then
    if [ $temperature -gt 70 ]; then
        echo "Perfect day for outdoor activities!"
    else
        echo "Sunny but a bit cold"
    fi
elif [ "$weather" = "rainy" ]; then
    if [ $temperature -gt 60 ]; then
        echo "Warm rain, good for plants"
    else
        echo "Cold and rainy, stay inside"
    fi
else
    echo "Weather is $weather"
fi
```

Short-Circuit Evaluation

Use `&&` and `||` for concise conditional execution:

```
#!/bin/bash
filename="test.txt"
```

```
# Execute command only if condition is true
[ -f "$filename" ] && echo "File exists"

# Execute command only if condition is false
[ ! -f "$filename" ] || echo "File exists"

# Chain multiple conditions
[ -f "$filename" ] && [ -r "$filename" ] && echo "File exists and is readable"

# Alternative to if-else
[ $# -eq 0 ] && echo "No arguments provided" || echo "Arguments provided"
```

Common Pitfalls and Best Practices

1. Always Quote Variables

```
# Wrong - can break with spaces
if [ $name = "John Doe" ]; then
    echo "Hello John"
fi

# Correct
if [ "$name" = "John Doe" ]; then
    echo "Hello John"
fi
```

2. Use [[ ]] for String Operations

```
# Better for pattern matching and regex
if [[ $filename == *.txt ]]; then
    echo "Text file"
fi

if [[ $email =~ ^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$ ]]; then
    echo "Valid email"
fi
```

3. Handle Empty Variables

```
# Safe way to handle potentially empty variables
if [ -n "${name:-}" ]; then
    echo "Name is set: $name"
else
    echo "Name is not set"
fi
```

4. Use Meaningful Conditions

```
# Good - readable and clear
if [ $user_age -ge $minimum_age ]; then
    grant_access
fi

# Bad - unclear magic numbers
if [ $age -ge 21 ]; then
    do_something
fi
```

5. Exit Codes for Error Handling

```
#!/bin/bash
if ! command_that_might_fail; then
    echo "Command failed"
    exit 1
fi

# Or check exit code explicitly
command_that_might_fail
if [ $? -ne 0 ]; then
    echo "Command failed with exit code $?"
    exit 1
fi
```

Conditional statements are the foundation of decision-making in shell scripts. Master these concepts to create scripts that can intelligently respond to different situations and handle various scenarios gracefully.

Loops

Loops allow you to execute a block of code repeatedly, making them essential for automating repetitive tasks. Bash provides several types of loops, each suited for different scenarios.

for Loops

The `for` loop is the most commonly used loop in Bash scripting.

Basic for Loop Syntax

```
for variable in list; do
    # commands
done
```

Simple Examples

```
#!/bin/bash

# Loop through a list of items
for fruit in apple banana orange; do
    echo "I like $fruit"
done

# Loop through numbers
for number in 1 2 3 4 5; do
    echo "Number: $number"
done
```

Looping Through Files

```
#!/bin/bash

# Process all .txt files in current directory
for file in *.txt; do
    if [ -f "$file" ]; then
```

```
        echo "Processing: $file"
        wc -l "$file"
    fi
done

# Process files in a specific directory
for file in /var/log/*.log; do
    if [ -f "$file" ]; then
        echo "Log file: $(basename "$file")"
        echo "Size: $(du -h "$file" | cut -f1)"
    fi
done
```

C-style for Loop

```
#!/bin/bash

# C-style for loop
for ((i=1; i<=10; i++)); do
    echo "Iteration: $i"
done

# Counting backwards
for ((i=10; i>=1; i--)); do
    echo "Countdown: $i"
done

# Step by 2
for ((i=0; i<=20; i+=2)); do
    echo "Even number: $i"
done
```

Using Brace Expansion

```
#!/bin/bash

# Loop through a range of numbers
for i in {1..10}; do
    echo "Number: $i"
done

# Loop with step
for i in {1..20..2}; do
    echo "Odd number: $i"
```

```
done

# Loop through letters
for letter in {a..z}; do
    echo "Letter: $letter"
done
```

Looping Through Arrays

```
#!/bin/bash

# Define an array
colors=("red" "green" "blue" "yellow")

# Loop through array elements
for color in "${colors[@]}"; do
    echo "Color: $color"
done

# Loop through array indices
for i in "${!colors[@]}"; do
    echo "Index $i: ${colors[i]}"
done
```

while Loops

The `while` loop continues executing as long as a condition is true.

Basic while Loop

```
#!/bin/bash

counter=1
while [ $counter -le 5 ]; do
    echo "Counter: $counter"
    ((counter++))
done
```

Reading File Line by Line

```
#!/bin/bash

filename="data.txt"

# Create sample file
cat > "$filename" << EOF
Line 1
Line 2
Line 3
EOF

# Read file line by line
while IFS= read -r line; do
    echo "Processing: $line"
done < "$filename"
```

Menu System with while Loop

```
#!/bin/bash

show_menu() {
    echo "=== Menu ==="
    echo "1. Show date"
    echo "2. Show users"
    echo "3. Show disk usage"
    echo "4. Exit"
    echo -n "Choose option: "
}

while true; do
    show_menu
    read choice

    case $choice in
        1)
            echo "Current date: $(date)"
            ;;
        2)
            echo "Logged in users:"
            who
            ;;
        3)
            echo "Disk usage:"
```

```
            df -h
            ;;
        4)
            echo "Goodbye!"
            break
            ;;
        *)
            echo "Invalid option. Please try again."
            ;;
    esac

    echo
    read -p "Press Enter to continue..."
    echo
done
```

Monitoring with while Loop

```
#!/bin/bash

# Monitor system load
echo "Monitoring system load (Ctrl+C to stop)..."

while true; do
    load=$(uptime | awk -F'load average:' '{print $2}' | awk '{print $1}' | sed 's/,//')
    echo "$(date): Load average: $load"

    # Alert if load is high
    if (( $(echo "$load > 2.0" | bc -l) )); then
        echo "WARNING: High system load!"
    fi

    sleep 5
done
```

until Loops

The `until` loop continues executing until a condition becomes true (opposite of while).

Basic until Loop

```
#!/bin/bash

counter=1
until [ $counter -gt 5 ]; do
    echo "Counter: $counter"
    ((counter++))
done
```

Waiting for a Service

```
#!/bin/bash

service_name="nginx"

echo "Waiting for $service_name to start..."
until systemctl is-active --quiet "$service_name"; do
    echo "Service not ready, waiting..."
    sleep 2
done

echo "$service_name is now running!"
```

Waiting for File

```
#!/bin/bash

target_file="/tmp/ready.flag"

echo "Waiting for file $target_file to appear..."
until [ -f "$target_file" ]; do
    echo "File not found, waiting..."
    sleep 1
done

echo "File found! Continuing..."
```

Loop Control Statements

break Statement

Exits the loop immediately:

```
#!/bin/bash

# Find first .txt file
for file in *; do
    if [[ "$file" == *.txt ]]; then
        echo "Found text file: $file"
        break
    fi
    echo "Checking: $file"
done
```

continue Statement

Skips the rest of the current iteration:

```
#!/bin/bash

# Process only .txt files
for file in *; do
    # Skip non-txt files
    if [[ "$file" != *.txt ]]; then
        continue
    fi

    echo "Processing text file: $file"
    wc -l "$file"
done
```

Nested Loops with break and continue

```
#!/bin/bash

# Nested loops example
for i in {1..3}; do
    echo "Outer loop: $i"

    for j in {1..5}; do
        if [ $j -eq 3 ]; then
```

```
            echo "  Skipping inner loop iteration $j"
            continue
        fi

        if [ $i -eq 2 ] && [ $j -eq 4 ]; then
            echo "  Breaking inner loop at $i,$j"
            break
        fi

        echo "  Inner loop: $j"
    done
done
```

Practical Loop Examples

Batch File Processing

```
#!/bin/bash

# Batch resize images
image_dir="./images"
output_dir="./thumbnails"

# Create output directory
mkdir -p "$output_dir"

for image in "$image_dir"/*.{jpg,jpeg,png}; do
    # Skip if no files match
    [ ! -f "$image" ] && continue

    filename=$(basename "$image")
    name="${filename%.*}"
    extension="${filename##*.}"

    echo "Processing: $filename"

    # Resize image (requires ImageMagick)
    if command -v convert >/dev/null 2>&1; then
        convert "$image" -resize 200x200 "$output_dir/${name}_thumb.$extension"
        echo "Created thumbnail: ${name}_thumb.$extension"
    else
        echo "ImageMagick not installed, skipping resize"
    fi
done
```

Log Analysis

```
#!/bin/bash

log_file="/var/log/apache2/access.log"
declare -A ip_count

# Count IP addresses in log file
while IFS= read -r line; do
    # Extract IP address (first field)
    ip=$(echo "$line" | awk '{print $1}')

    # Skip invalid IPs
    if [[ $ip =~ ^[0-9]+\.[0-9]+\.[0-9]+\.[0-9]+$ ]]; then
        ((ip_count[$ip]++))
    fi
done < "$log_file"

# Display top 10 IPs
echo "Top 10 IP addresses:"
for ip in "${!ip_count[@]}"; do
    echo "$ip: ${ip_count[$ip]}"
done | sort -k2 -nr | head -10
```

System Backup Script

```
#!/bin/bash

# Directories to backup
backup_dirs=("/home/user/documents" "/home/user/projects" "/etc")
backup_destination="/backup/$(date +%Y%m%d)"
log_file="/var/log/backup.log"

# Create backup destination
mkdir -p "$backup_destination"

# Function to log messages
log_message() {
    echo "$(date '+%Y-%m-%d %H:%M:%S') - $1" | tee -a "$log_file"
}

log_message "Starting backup process"

# Backup each directory
for dir in "${backup_dirs[@]}"; do
```

```
    if [ -d "$dir" ]; then
        dir_name=$(basename "$dir")
        log_message "Backing up $dir"

        if tar -czf "$backup_destination/${dir_name}.tar.gz" "$dir"; then
            log_message "Successfully backed up $dir"
        else
            log_message "ERROR: Failed to backup $dir"
        fi
    else
        log_message "WARNING: Directory $dir does not exist"
    fi
done

log_message "Backup process completed"

# Clean up old backups (keep only last 7 days)
find /backup -name "20*" -type d -mtime +7 -exec rm -rf {} \;
log_message "Cleaned up old backups"
```

User Management Script

```
#!/bin/bash

# Create multiple users from a list
users_file="users.txt"

# Create sample users file
cat > "$users_file" << EOF
john:John Doe:developer
jane:Jane Smith:admin
bob:Bob Johnson:user
EOF

# Process each user
while IFS=':' read -r username fullname role; do
    # Skip empty lines and comments
    [[ -z "$username" || "$username" =~ ^# ]] && continue

    echo "Processing user: $username"

    # Check if user already exists
    if id "$username" >/dev/null 2>&1; then
        echo "User $username already exists, skipping"
        continue
```

```
    fi

    # Create user
    if useradd -m -c "$fullname" "$username"; then
        echo "Created user: $username ($fullname)"

        # Set initial password
        echo "$username:temp123" | chpasswd
        echo "Set temporary password for $username"

        # Add to appropriate group based on role
        case "$role" in
            "admin")
                usermod -aG sudo "$username"
                echo "Added $username to sudo group"
                ;;
            "developer")
                usermod -aG developers "$username" 2>/dev/null || echo "developers group not
                ;;
        esac
    else
        echo "Failed to create user: $username"
    fi

done < "$users_file"
```

Loop Performance Tips

Avoid Unnecessary Command Substitution

```
# Slow - calls external command in each iteration
for i in {1..1000}; do
    date_str=$(date)
    echo "$i: $date_str"
done

# Fast - call command once
date_str=$(date)
for i in {1..1000}; do
    echo "$i: $date_str"
done
```

Use Built-in Commands When Possible

```
# Slow - external command
for file in *.txt; do
    lines=$(wc -l < "$file")
    echo "$file: $lines lines"
done

# Faster - but still external command per file
for file in *.txt; do
    while IFS= read -r line; do
        ((count++))
    done < "$file"
    echo "$file: $count lines"
    count=0
done
```

Process Files Efficiently

```
# Process multiple files with one command
find . -name "*.log" -print0 | while IFS= read -r -d '' file; do
    echo "Processing: $file"
    # Process file
done
```

Common Loop Patterns

Retry Pattern

```
retry_command() {
    local max_attempts=3
    local attempt=1

    while [ $attempt -le $max_attempts ]; do
        echo "Attempt $attempt of $max_attempts"

        if command_that_might_fail; then
            echo "Command succeeded"
            return 0
        fi

        echo "Command failed, retrying..."
        ((attempt++))
```

```
        sleep 2
    done

    echo "Command failed after $max_attempts attempts"
    return 1
}
```

Progress Indicator

```
#!/bin/bash

total_items=100

for ((i=1; i<=total_items; i++)); do
    # Simulate work
    sleep 0.1

    # Calculate progress
    progress=$((i * 100 / total_items))

    # Display progress bar
    printf "\rProgress: [%-50s] %d%%" $(printf "#%.0s" $(seq 1 $((progress/2)))) $progress
done

echo
echo "Complete!"
```

Loops are powerful constructs that enable automation and batch processing in your shell scripts. Understanding when and how to use each type of loop will help you write more efficient and effective scripts.

Functions

Function Basics

Functions are reusable blocks of code that help organize your scripts, reduce repetition, and make your code more maintainable. They allow you to break complex scripts into smaller, manageable pieces.

What are Functions?

Functions in Bash are named blocks of code that can be called multiple times throughout your script. They help you:

- **Organize code**: Break large scripts into logical pieces
- **Reduce repetition**: Write once, use many times
- **Improve readability**: Make scripts easier to understand
- **Enable testing**: Test individual components
- **Facilitate maintenance**: Update code in one place

Function Syntax

Method 1: Using the `function` keyword

```
function function_name() {
    # function body
    commands
}
```

Method 2: POSIX-compatible syntax (preferred)

```
function_name() {
    # function body
    commands
}
```

Basic Function Examples

Simple Function

```
#!/bin/bash

# Define a function
greet() {
    echo "Hello, World!"
}

# Call the function
greet
```

Function with Commands

```
#!/bin/bash

show_date() {
    echo "Current date and time:"
    date
    echo "Uptime:"
    uptime
}

# Call the function
show_date
```

Function Parameters

Functions can accept parameters (arguments) to make them more flexible:

Accessing Parameters

```
#!/bin/bash

greet_user() {
    echo "Hello, $1!"
    echo "Welcome to $2"
}

# Call function with arguments
```

```
greet_user "Alice" "Bash scripting"
```

Multiple Parameters

```
#!/bin/bash

user_info() {
    local name="$1"
    local age="$2"
    local city="$3"

    echo "Name: $name"
    echo "Age: $age"
    echo "City: $city"
}

# Call with multiple arguments
user_info "John Doe" "30" "New York"
```

Parameter Variables

```
#!/bin/bash

show_params() {
    echo "Function name: $0"
    echo "First parameter: $1"
    echo "Second parameter: $2"
    echo "All parameters: $@"
    echo "Number of parameters: $#"
    echo "All parameters as single string: $*"
}

show_params "arg1" "arg2" "arg3"
```

Return Values

Functions can return values using the `return` statement:

Return Exit Status

```
#!/bin/bash

check_file() {
    local filename="$1"

    if [ -f "$filename" ]; then
        echo "File $filename exists"
        return 0  # Success
    else
        echo "File $filename does not exist"
        return 1  # Failure
    fi
}

# Use function and check return value
if check_file "test.txt"; then
    echo "File check passed"
else
    echo "File check failed"
fi
```

Return Values via Echo

```
#!/bin/bash

get_file_size() {
    local filename="$1"

    if [ -f "$filename" ]; then
        stat -f%z "$filename" 2>/dev/null || stat -c%s "$filename" 2>/dev/null
    else
        echo "0"
    fi
}

# Capture function output
file_size=$(get_file_size "test.txt")
echo "File size: $file_size bytes"
```

Local Variables

Use `local` to create variables that exist only within the function:

Without Local Variables (Global)

```
#!/bin/bash

global_example() {
    name="Alice"  # This modifies global variable
    echo "Inside function: $name"
}

name="Bob"
echo "Before function: $name"
global_example
echo "After function: $name"  # Changed to Alice
```

With Local Variables

```
#!/bin/bash

local_example() {
    local name="Alice"  # This is local to the function
    echo "Inside function: $name"
}

name="Bob"
echo "Before function: $name"
local_example
echo "After function: $name"  # Still Bob
```

Best Practice Example

```
#!/bin/bash

calculate_area() {
    local length="$1"
    local width="$2"
    local area

    area=$((length * width))
    echo "$area"
}

# Use the function
room_area=$(calculate_area 10 12)
```

```
echo "Room area: $room_area square feet"
```

Practical Function Examples

File Operations Function

```
#!/bin/bash

backup_file() {
    local source_file="$1"
    local backup_dir="${2:-./backup}"

    # Validate input
    if [ -z "$source_file" ]; then
        echo "Error: Source file not specified"
        return 1
    fi

    if [ ! -f "$source_file" ]; then
        echo "Error: Source file '$source_file' does not exist"
        return 1
    fi

    # Create backup directory if it doesn't exist
    mkdir -p "$backup_dir"

    # Create backup with timestamp
    local backup_name="$(basename "$source_file").backup.$(date +%Y%m%d_%H%M%S)"

    if cp "$source_file" "$backup_dir/$backup_name"; then
        echo "Backup created: $backup_dir/$backup_name"
        return 0
    else
        echo "Error: Failed to create backup"
        return 1
    fi
}

# Use the function
backup_file "/etc/hosts"
backup_file "important.txt" "/safe/location"
```

System Information Function

```
#!/bin/bash

show_system_info() {
    local info_type="$1"

    case "$info_type" in
        "cpu")
            echo "CPU Information:"
            lscpu | grep "Model name" || echo "CPU info not available"
            ;;
        "memory")
            echo "Memory Information:"
            free -h
            ;;
        "disk")
            echo "Disk Usage:"
            df -h
            ;;
        "all")
            show_system_info "cpu"
            echo
            show_system_info "memory"
            echo
            show_system_info "disk"
            ;;
        *)
            echo "Usage: show_system_info {cpu|memory|disk|all}"
            return 1
            ;;
    esac
}

# Use the function
show_system_info "all"
```

String Processing Function

```
#!/bin/bash

process_string() {
    local input="$1"
    local operation="$2"
```

```
    case "$operation" in
        "upper")
            echo "${input^^}"
            ;;
        "lower")
            echo "${input,,}"
            ;;
        "length")
            echo "${#input}"
            ;;
        "reverse")
            echo "$input" | rev
            ;;
        "words")
            echo "$input" | wc -w
            ;;
        *)
            echo "Usage: process_string <string> {upper|lower|length|reverse|words}"
            return 1
            ;;
    esac
}

# Use the function
text="Hello World"
echo "Original: $text"
echo "Uppercase: $(process_string "$text" "upper")"
echo "Lowercase: $(process_string "$text" "lower")"
echo "Length: $(process_string "$text" "length")"
echo "Reversed: $(process_string "$text" "reverse")"
echo "Word count: $(process_string "$text" "words")"
```

Function Libraries

You can create reusable function libraries:

Create a Library File (utils.sh)

```
#!/bin/bash
# utils.sh - Utility functions library

# Logging function
log() {
    local level="$1"
```

```
    shift
    local message="$*"
    echo "$(date '+%Y-%m-%d %H:%M:%S') [$level] $message"
}

# Check if command exists
command_exists() {
    command -v "$1" >/dev/null 2>&1
}

# Validate email format
is_valid_email() {
    local email="$1"
    [[ $email =~ ^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$ ]]
}

# Get file extension
get_extension() {
    local filename="$1"
    echo "${filename##*.}"
}

# Create directory if it doesn't exist
ensure_directory() {
    local dir="$1"
    if [ ! -d "$dir" ]; then
        mkdir -p "$dir"
        log "INFO" "Created directory: $dir"
    fi
}
```

Use the Library in Your Script

```
#!/bin/bash
# main_script.sh

# Source the utility functions
source ./utils.sh

# Use the functions
log "INFO" "Script started"

if command_exists "git"; then
    log "INFO" "Git is available"
else
```

```
    log "ERROR" "Git is not installed"
    exit 1
fi

email="user@example.com"
if is_valid_email "$email"; then
    log "INFO" "Email format is valid"
else
    log "ERROR" "Invalid email format"
fi

filename="document.pdf"
extension=$(get_extension "$filename")
log "INFO" "File extension: $extension"

ensure_directory "/tmp/myapp"
log "INFO" "Script completed"
```

Function Best Practices

1. Use Descriptive Names

```
# Good
calculate_monthly_payment() {
    # function body
}

# Bad
calc() {
    # function body
}
```

2. Validate Input Parameters

```
backup_database() {
    local db_name="$1"
    local backup_path="$2"

    # Validate required parameters
    if [ -z "$db_name" ]; then
        echo "Error: Database name is required"
        return 1
    fi
```

```
    if [ -z "$backup_path" ]; then
        echo "Error: Backup path is required"
        return 1
    fi

    # Function logic here
}
```

3. Use Local Variables

```
process_data() {
    local input_file="$1"
    local output_file="$2"
    local temp_file
    local line_count

    # All variables are local to this function
    temp_file=$(mktemp)
    line_count=$(wc -l < "$input_file")

    # Process data...
}
```

4. Provide Usage Information

```
convert_temperature() {
    local temp="$1"
    local from_unit="$2"
    local to_unit="$3"

    if [ $# -ne 3 ]; then
        echo "Usage: convert_temperature <temperature> <from_unit> <to_unit>"
        echo "Units: C (Celsius), F (Fahrenheit), K (Kelvin)"
        return 1
    fi

    # Conversion logic here
}
```

5. Handle Errors Gracefully

```
download_file() {
    local url="$1"
    local destination="$2"

    if ! command -v curl >/dev/null 2>&1; then
        echo "Error: curl is not installed"
        return 1
    fi

    if ! curl -o "$destination" "$url"; then
        echo "Error: Failed to download $url"
        return 1
    fi

    echo "Successfully downloaded to $destination"
    return 0
}
```

Common Function Patterns

Configuration Function

```
load_config() {
    local config_file="${1:-config.conf}"

    if [ -f "$config_file" ]; then
        source "$config_file"
        echo "Configuration loaded from $config_file"
    else
        echo "Warning: Configuration file $config_file not found"
        echo "Using default settings"
    fi
}
```

Cleanup Function

```
cleanup() {
    local temp_files=("$@")

    echo "Cleaning up temporary files..."
    for file in "${temp_files[@]}"; do
```

```
        if [ -f "$file" ]; then
            rm "$file"
            echo "Removed: $file"
        fi
    done
}

# Set trap to call cleanup on exit
trap 'cleanup "$temp_file1" "$temp_file2"' EXIT
```

Menu Function

```
show_menu() {
    echo "=== Main Menu ==="
    echo "1. Option 1"
    echo "2. Option 2"
    echo "3. Option 3"
    echo "4. Exit"
    echo -n "Choose an option: "
}

handle_choice() {
    local choice="$1"

    case "$choice" in
        1) echo "You chose option 1" ;;
        2) echo "You chose option 2" ;;
        3) echo "You chose option 3" ;;
        4) echo "Goodbye!"; exit 0 ;;
        *) echo "Invalid choice" ;;
    esac
}
```

Functions are powerful tools that make your Bash scripts more organized, maintainable, and reusable. They're essential for writing professional-quality shell scripts that are easy to understand and modify.

Misc

Loop from 1 to 5

Loops are one of the core building blocks in Bash scripting. They let you run a set of commands repeatedly, which is useful for anything from processing files in a directory to automating system tasks. Among the different types of loops, the for loop is one of the most commonly used. It lets you iterate over a sequence of values, such as numbers, strings, or items in an array.

In this guide, you'll learn how Bash for loops work. We'll go over the standard syntax, the C-style variant, and how to use break and continue to control loop execution. You'll also see how nested loops work, with plenty of examples along the way to show how you can use them in real scripts.

Standard Bash for Loop

The standard Bash for loop has the following syntax:

for variable in list do commands done Here, variable is a temporary variable that takes on each value from the list in turn, and the commands are executed for each iteration. The list can be a list of values separated by spaces, strings separated by spaces, a range of numbers, or even the output of a command.

Loop Over Number Ranges

One common use case for the for loop is to iterate over a range of numbers. This can be achieved using the seq command, which generates a sequence of numbers:

for i in $(seq 1 5) do echo $i done Here's how it works:

The loop starts by assigning the first value from the sequence $(seq 1 5) to the variable i. The echo $i command within the loop body is then executed. The loop then moves to the next value in the sequence, assigns it to the variable, and executes the echo $i command again. This process continues until all values in the sequence have been iterated over. You can also specify an increment value for the sequence:

Loop from 1 to 10, incrementing by 2

for i in $(seq 1 2 10) do echo $i done Here is another common alternative syntax for specifying a range of numbers:

```
for i in {1..5}
do
  echo "Iteration $i"
done
```

The above syntax is more common in bash scripts. Starting from Bash 4, it's now possible to specify an increment when using ranges. Here is what it looks like:

```
# start..end..increment
for i in {1..10..2}
do
  echo "Iteration $i"
done
```

Loop Over Strings

The for loop can also iterate over a list of strings. Consider the following example:

Loop over a list of strings

```
for color in red green blue
do
  echo "The color is $color"
done
```

In this case, the loop iterates over the specified color strings, displaying each one in the output.

You can also use the for loop to iterate over characters in a string:

```
mystring="Hello, World!"
for char in $(echo "$mystring" | grep -o .)
do
    echo "$char"
done
```

In the above example:

The string “Hello, World!” is stored in the variable mystring. The grep -o . command matches and prints each character in the string individually. The for loop iterates over each character output by grep, assigning it to the variable char. Within the loop body, the current character is printed using the echo command. Loop Over Arrays

Bash arrays provide a convenient way to store multiple values. Let’s see how to use a for loop to iterate over the elements of an array:

Define an array

colors=(“red” “green” “blue”)

Loop over the array

```
for color in "${colors[@]}"
do
  echo "The color is $color"
done
```

Here we defined an array named colors and iterate over each element of the array echoing its value. Note the use of the [@] syntax to expand the array into a list of values.

If my Linux guides, cheat sheets, or deep dives have helped you learn or solve a problem, consider going paid. It keeps this work going and independent.

Upgrade to paid C-Style Bash for Loop

Bash also supports a C-style for loop, which provides more control over loop variables and conditions. The syntax for the C-style for loop is:

for ((variable assignment; condition; iteration process)) do commands done Here, variable assignment is executed once before the loop starts, condition is the condition that is evaluated before each iteration, and iteration process is executed after each iteration.

The following example prints numbers from 1 to 5.

Loop from 1 to 5

for ((i=1; i<=5; i++)) do echo $i done There are a couple of things you need to know about this syntax:

Variable value assignments can contain spaces. It's possible to omit the dollar sign for the variable in the condition. The equation for the iteration process doesn't use the expr command format to increment the value of the value. Break and Continue Statements

The break and continue statements can be used to control the flow of execution within a for loop, allowing you to skip or terminate iterations based on specific conditions. These statements can be particularly useful when dealing with complex or nested loops, or when you want to implement specific logic within your loop.

Break Statement

The break statement terminates the loop immediately and transfers control to the next statement following the loop. This can be useful in situations where you want to exit the loop early based on a certain condition, such as when you've found the desired value or when a specific condition is met.

Loop through an array and break when a specific value is found

```
fruits=("apple" "banana" "orange" "grape" "pear")

for fruit in "${fruits[@]}"
do
  if [ "$fruit" == "orange" ]
  then
    echo "Found orange!"
    break
  fi
  echo "Current fruit: $fruit"
done
```

In this example, the loop iterates over the fruits array, and when the value “orange” is encountered, the break statement is executed, terminating the loop and printing “Found orange!”.

Continue Statement

The continue statement skips the current iteration of the loop and moves to the next iteration. This can be useful when you want to skip certain iterations based on a specific condition, such as when dealing with invalid or unwanted data.

Loop through a range of numbers and skip multiples of 3

for i in $(seq 1 10) do if [$((i % 3)) -eq 0] then continue fi echo $i done In this example, the loop iterates over the numbers from 1 to 10, but it skips the multiples of 3 using the continue statement.

Both break and continue statements can be used in combination with conditional statements (if, case, etc.) to implement complex logic within your loops. However, it's important to use them judiciously and avoid excessive use, as this can make your code harder to read and maintain.

Nested For Loops

In Bash, it's possible to have one for loop nested inside another for loop. This construct is known as a nested loop, and it allows you to iterate over multidimensional data structures or perform operations that require nested iterations.

The syntax for a nested for loop is as follows:

for outer_variable in outer_list do for inner_variable in inner_list do commands done done Here, the outer loop iterates over theouter_list, and for each iteration of the outer loop, the inner loop iterates over the inner_list. The commands are executed for each combination of the outer and inner loop variables.

Consider the following example:

for i in $(seq 1 2) do for j in $(seq 1 3) do echo "Outer: $i, Inner: $j" done done Let's break down the code:

The outer loop iterates over the numbers 1 and 2, assigning each value to the variable i. For each iteration of the outer loop, the inner loop iterates over the numbers 1, 2, and 3, assigning each value to the variable j. Within the inner loop body, the current values of i and j are printed using the echo command. Thanks for reading!

If you enjoyed this content, don't forget to leave a like and subscribe to get more posts like this every week.

Workbook

Basic Bash Examples

This section contains fundamental Bash scripting examples to help you practice and understand core concepts.

Hello World Variations

Simple Hello World

```
#!/bin/bash
echo "Hello, World!"
```

Hello World with Variables

```
#!/bin/bash
name="World"
echo "Hello, $name!"
```

Interactive Hello World

```
#!/bin/bash
echo "What's your name?"
read name
echo "Hello, $name! Nice to meet you."
```

Hello World with Date

```
#!/bin/bash
echo "Hello, World!"
echo "Today is: $(date)"
echo "Current user: $(whoami)"
```

Variable Examples

Basic Variable Operations

```
#!/bin/bash
# Variable assignment
first_name="John"
last_name="Doe"
age=25

# Variable usage
echo "Name: $first_name $last_name"
echo "Age: $age"

# Variable concatenation
full_name="$first_name $last_name"
echo "Full name: $full_name"
```

Environment Variables

```
#!/bin/bash
echo "Your home directory: $HOME"
echo "Your username: $USER"
echo "Your shell: $SHELL"
echo "Current path: $PATH"
```

Variable Default Values

```
#!/bin/bash
# Set default values
database_host=${DB_HOST:-"localhost"}
database_port=${DB_PORT:-"5432"}
database_name=${DB_NAME:-"myapp"}

echo "Database configuration:"
echo "Host: $database_host"
echo "Port: $database_port"
echo "Database: $database_name"
```

Variable Length and Manipulation

```
#!/bin/bash
text="Hello World"

echo "Original text: $text"
echo "Length: ${#text}"
echo "Uppercase: ${text^^}"
echo "Lowercase: ${text,,}"
echo "First 5 characters: ${text:0:5}"
echo "Last 5 characters: ${text: -5}"
```

Command Line Arguments

Basic Argument Handling

```
#!/bin/bash
echo "Script name: $0"
echo "First argument: $1"
echo "Second argument: $2"
echo "All arguments: $@"
echo "Number of arguments: $#"
```

Argument Validation

```
#!/bin/bash
if [ $# -eq 0 ]; then
    echo "Usage: $0 <name> [age]"
    exit 1
fi

name="$1"
age="${2:-unknown}"

echo "Name: $name"
echo "Age: $age"
```

Processing Multiple Arguments

```
#!/bin/bash
echo "Processing all arguments:"
for arg in "$@"; do
    echo "Argument: $arg"
done
```

Simple Calculations

Basic Arithmetic

```
#!/bin/bash
a=10
b=5

echo "a = $a, b = $b"
echo "Addition: $((a + b))"
echo "Subtraction: $((a - b))"
echo "Multiplication: $((a * b))"
echo "Division: $((a / b))"
echo "Modulo: $((a % b))"
```

Calculator Script

```
#!/bin/bash
echo "Simple Calculator"
read -p "Enter first number: " num1
read -p "Enter operator (+, -, *, /): " op
read -p "Enter second number: " num2

case $op in
    +) result=$((num1 + num2)) ;;
    -) result=$((num1 - num2)) ;;
    \*) result=$((num1 * num2)) ;;
    /)
        if [ $num2 -eq 0 ]; then
            echo "Error: Division by zero"
            exit 1
        fi
        result=$((num1 / num2))
        ;;
    *) echo "Invalid operator"; exit 1 ;;
```

```
esac

echo "Result: $num1 $op $num2 = $result"
```

Temperature Converter

```
#!/bin/bash
echo "Temperature Converter"
echo "1. Celsius to Fahrenheit"
echo "2. Fahrenheit to Celsius"
read -p "Choose option (1 or 2): " choice

case $choice in
    1)
        read -p "Enter temperature in Celsius: " celsius
        fahrenheit=$(echo "scale=2; $celsius * 9/5 + 32" | bc)
        echo "$celsius°C = $fahrenheit°F"
        ;;
    2)
        read -p "Enter temperature in Fahrenheit: " fahrenheit
        celsius=$(echo "scale=2; ($fahrenheit - 32) * 5/9" | bc)
        echo "$fahrenheit°F = $celsius°C"
        ;;
    *)
        echo "Invalid choice"
        exit 1
        ;;
esac
```

File Operations

File Information

```
#!/bin/bash
filename="$1"

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

if [ -f "$filename" ]; then
    echo "File: $filename"
```

```
    echo "Size: $(stat -c%s "$filename") bytes"
    echo "Last modified: $(stat -c%y "$filename")"
    echo "Permissions: $(stat -c%A "$filename")"
    echo "Owner: $(stat -c%U "$filename")"
    echo "Group: $(stat -c%G "$filename")"
else
    echo "File '$filename' does not exist"
fi
```

File Backup

```
#!/bin/bash
source_file="$1"

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

if [ ! -f "$source_file" ]; then
    echo "Error: File '$source_file' does not exist"
    exit 1
fi

backup_file="${source_file}.backup.$(date +%Y%m%d_%H%M%S)"

if cp "$source_file" "$backup_file"; then
    echo "Backup created: $backup_file"
else
    echo "Error: Failed to create backup"
    exit 1
fi
```

Directory Listing

```
#!/bin/bash
directory="${1:-.}"

echo "Contents of directory: $directory"
echo "================================="

if [ -d "$directory" ]; then
    for item in "$directory"/*; do
        if [ -f "$item" ]; then
```

```
            echo "FILE: $(basename "$item")"
        elif [ -d "$item" ]; then
            echo "DIR:  $(basename "$item")"
        fi
    done
else
    echo "Error: '$directory' is not a directory"
fi
```

Text Processing

Word Count

```
#!/bin/bash
filename="$1"

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

if [ -f "$filename" ]; then
    lines=$(wc -l < "$filename")
    words=$(wc -w < "$filename")
    chars=$(wc -c < "$filename")

    echo "File: $filename"
    echo "Lines: $lines"
    echo "Words: $words"
    echo "Characters: $chars"
else
    echo "File '$filename' does not exist"
fi
```

Text Search

```
#!/bin/bash
if [ $# -ne 2 ]; then
    echo "Usage: $0 <search_term> <filename>"
    exit 1
fi

search_term="$1"
```

```
filename="$2"

if [ ! -f "$filename" ]; then
    echo "File '$filename' does not exist"
    exit 1
fi

echo "Searching for '$search_term' in '$filename':"
grep -n "$search_term" "$filename"

count=$(grep -c "$search_term" "$filename")
echo "Found $count occurrences"
```

Line Numbering

```
#!/bin/bash
filename="$1"

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

if [ -f "$filename" ]; then
    line_number=1
    while IFS= read -r line; do
        printf "%3d: %s\n" $line_number "$line"
        ((line_number++))
    done < "$filename"
else
    echo "File '$filename' does not exist"
fi
```

System Information

System Status

```
#!/bin/bash
echo "=== System Information ==="
echo "Hostname: $(hostname)"
echo "Uptime: $(uptime -p)"
echo "Current user: $(whoami)"
echo "Current directory: $(pwd)"
```

```
echo "Date: $(date)"
echo

echo "=== Memory Usage ==="
free -h

echo
echo "=== Disk Usage ==="
df -h

echo
echo "=== CPU Information ==="
lscpu | grep "Model name"
```

Process Monitor

```
#!/bin/bash
echo "Top 10 processes by CPU usage:"
ps aux --sort=-%cpu | head -11

echo
echo "Top 10 processes by memory usage:"
ps aux --sort=-%mem | head -11
```

Network Information

```
#!/bin/bash
echo "=== Network Information ==="
echo "Hostname: $(hostname)"
echo "Domain: $(hostname -d 2>/dev/null || echo "Not set")"

echo
echo "=== Network Interfaces ==="
ip addr show | grep -E "^[0-9]+:|inet "

echo
echo "=== Routing Table ==="
ip route show
```

User Input Examples

Menu System

```
#!/bin/bash
while true; do
    echo
    echo "=== Main Menu ==="
    echo "1. Show date and time"
    echo "2. Show current directory"
    echo "3. List files"
    echo "4. Show disk usage"
    echo "5. Exit"
    echo
    read -p "Choose an option (1-5): " choice

    case $choice in
        1) date ;;
        2) pwd ;;
        3) ls -la ;;
        4) df -h ;;
        5) echo "Goodbye!"; exit 0 ;;
        *) echo "Invalid choice. Please try again." ;;
    esac

    read -p "Press Enter to continue..."
done
```

User Registration

```
#!/bin/bash
echo "User Registration Form"
echo "======================"

read -p "First Name: " first_name
read -p "Last Name: " last_name
read -p "Email: " email
read -p "Age: " age
read -s -p "Password: " password
echo

echo
echo "Registration Summary:"
echo "Name: $first_name $last_name"
echo "Email: $email"
```

```
echo "Age: $age"
echo "Password: [hidden]"

read -p "Is this information correct? (y/n): " confirm
if [ "$confirm" = "y" ] || [ "$confirm" = "Y" ]; then
    echo "Registration completed!"
else
    echo "Registration cancelled."
fi
```

Quiz Game

```
#!/bin/bash
score=0
total_questions=3

echo "Welcome to the Bash Quiz!"
echo "========================"

# Question 1
echo "Question 1: What command shows the current directory?"
echo "a) pwd"
echo "b) cd"
echo "c) ls"
read -p "Your answer: " answer1

if [ "$answer1" = "a" ] || [ "$answer1" = "A" ]; then
    echo "Correct!"
    ((score++))
else
    echo "Wrong! The correct answer is 'a) pwd'"
fi

# Question 2
echo
echo "Question 2: What symbol is used for comments in bash?"
echo "a) //"
echo "b) #"
echo "c) /*"
read -p "Your answer: " answer2

if [ "$answer2" = "b" ] || [ "$answer2" = "B" ]; then
    echo "Correct!"
    ((score++))
else
```

```
    echo "Wrong! The correct answer is 'b) #'"
fi

# Question 3
echo
echo "Question 3: What command lists files and directories?"
echo "a) list"
echo "b) dir"
echo "c) ls"
read -p "Your answer: " answer3

if [ "$answer3" = "c" ] || [ "$answer3" = "C" ]; then
    echo "Correct!"
    ((score++))
else
    echo "Wrong! The correct answer is 'c) ls'"
fi

echo
echo "Quiz completed!"
echo "Your score: $score out of $total_questions"

percentage=$((score * 100 / total_questions))
if [ $percentage -ge 80 ]; then
    echo "Excellent work!"
elif [ $percentage -ge 60 ]; then
    echo "Good job!"
else
    echo "Keep practicing!"
fi
```

String Manipulation

String Operations

```
#!/bin/bash
text="Hello World Programming"

echo "Original: $text"
echo "Length: ${#text}"
echo "Uppercase: ${text^^}"
echo "Lowercase: ${text,,}"
echo "First word: ${text% *}"
echo "Last word: ${text##* }"
echo "Remove 'Hello': ${text/Hello/}"
```

```
echo "Replace 'World' with 'Bash': ${text/World/Bash}"
echo "First 10 chars: ${text:0:10}"
echo "From position 6: ${text:6}"
```

Palindrome Checker

```
#!/bin/bash
read -p "Enter a word to check if it's a palindrome: " word

# Convert to lowercase and remove spaces
clean_word=$(echo "$word" | tr '[:upper:]' '[:lower:]' | tr -d ' ')

# Reverse the string
reversed=$(echo "$clean_word" | rev)

if [ "$clean_word" = "$reversed" ]; then
    echo "'$word' is a palindrome!"
else
    echo "'$word' is not a palindrome."
fi
```

Password Generator

```
#!/bin/bash
read -p "Enter password length (default 12): " length
length=${length:-12}

# Generate password using /dev/urandom
password=$(tr -dc 'A-Za-z0-9!@#$%^&*' < /dev/urandom | head -c "$length")

echo "Generated password: $password"
echo "Password strength:"
echo "Length: $length characters"

# Check password strength
if [[ $password =~ [A-Z] ]]; then
    echo "  Contains uppercase letters"
fi

if [[ $password =~ [a-z] ]]; then
    echo "  Contains lowercase letters"
fi

if [[ $password =~ [0-9] ]]; then
```

```
    echo "  Contains numbers"
fi

if [[ $password =~ [!@#\$%\^&\*] ]]; then
    echo "  Contains special characters"
fi
```

Array Examples

Basic Array Operations

```
#!/bin/bash
# Create array
fruits=("apple" "banana" "orange" "grape")

echo "All fruits: ${fruits[@]}"
echo "Number of fruits: ${#fruits[@]}"
echo "First fruit: ${fruits[0]}"
echo "Last fruit: ${fruits[-1]}"

# Add element
fruits+=("mango")
echo "After adding mango: ${fruits[@]}"

# Loop through array
echo "Fruits list:"
for i in "${!fruits[@]}"; do
    echo "$((i+1)). ${fruits[i]}"
done
```

Shopping List

```
#!/bin/bash
declare -a shopping_list

echo "Shopping List Manager"
echo "====================="

while true; do
    echo
    echo "1. Add item"
    echo "2. Remove item"
    echo "3. Show list"
```

```
echo "4. Clear list"
echo "5. Exit"
read -p "Choose option: " choice

case $choice in
    1)
        read -p "Enter item to add: " item
        shopping_list+=("$item")
        echo "Added '$item' to the list"
        ;;
    2)
        if [ ${#shopping_list[@]} -eq 0 ]; then
            echo "List is empty"
        else
            echo "Current items:"
            for i in "${!shopping_list[@]}"; do
                echo "$((i+1)). ${shopping_list[i]}"
            done
            read -p "Enter item number to remove: " num
            if [ "$num" -ge 1 ] && [ "$num" -le ${#shopping_list[@]} ]; then
                removed_item="${shopping_list[$((num-1))]}"
                unset shopping_list[$((num-1))]
                shopping_list=("${shopping_list[@]}")  # Re-index array
                echo "Removed '$removed_item'"
            else
                echo "Invalid item number"
            fi
        fi
        ;;
    3)
        if [ ${#shopping_list[@]} -eq 0 ]; then
            echo "Shopping list is empty"
        else
            echo "Shopping List:"
            for i in "${!shopping_list[@]}"; do
                echo "$((i+1)). ${shopping_list[i]}"
            done
        fi
        ;;
    4)
        shopping_list=()
        echo "List cleared"
        ;;
    5)
        echo "Goodbye!"
        exit 0
        ;;
```

```
        *)
            echo "Invalid choice"
            ;;
    esac
done
```

Grade Calculator

```
#!/bin/bash
declare -a grades
declare -a subjects

echo "Grade Calculator"
echo "================"

read -p "How many subjects? " num_subjects

for ((i=1; i<=num_subjects; i++)); do
    read -p "Enter subject $i name: " subject
    read -p "Enter grade for $subject: " grade

    subjects+=("$subject")
    grades+=("$grade")
done

# Calculate average
total=0
for grade in "${grades[@]}"; do
    total=$((total + grade))
done

average=$((total / num_subjects))

echo
echo "Grade Report:"
echo "============="
for i in "${!subjects[@]}"; do
    echo "${subjects[i]}: ${grades[i]}"
done

echo
echo "Average: $average"

if [ $average -ge 90 ]; then
    echo "Grade: A (Excellent)"
```

```
elif [ $average -ge 80 ]; then
    echo "Grade: B (Good)"
elif [ $average -ge 70 ]; then
    echo "Grade: C (Average)"
elif [ $average -ge 60 ]; then
    echo "Grade: D (Below Average)"
else
    echo "Grade: F (Fail)"
fi
```

Intermediate Bash Examples

This section contains more advanced Bash scripting examples that demonstrate intermediate concepts and real-world applications.

Function Examples

Library of Utility Functions

```
#!/bin/bash
# utils.sh - Utility functions library

# Logging function with levels
log() {
    local level="$1"
    shift
    local message="$*"
    local timestamp=$(date '+%Y-%m-%d %H:%M:%S')

    case "$level" in
        ERROR) echo "[$timestamp] ERROR: $message" >&2 ;;
        WARN)  echo "[$timestamp] WARN:  $message" ;;
        INFO)  echo "[$timestamp] INFO:  $message" ;;
        DEBUG) [ "$DEBUG" = "1" ] && echo "[$timestamp] DEBUG: $message" ;;
    esac
}

# Check if command exists
command_exists() {
    command -v "$1" >/dev/null 2>&1
}

# Validate email format
validate_email() {
    local email="$1"
    [[ $email =~ ^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$ ]]
}

# Get file extension
```

```
get_extension() {
    local filename="$1"
    echo "${filename##*.}"
}

# Convert bytes to human readable format
bytes_to_human() {
    local bytes="$1"
    local units=("B" "KB" "MB" "GB" "TB")
    local unit=0

    while [ $bytes -gt 1024 ] && [ $unit -lt 4 ]; do
        bytes=$((bytes / 1024))
        ((unit++))
    done

    echo "$bytes ${units[$unit]}"
}

# Create backup with timestamp
backup_file() {
    local file="$1"
    local backup_dir="${2:-./backups}"

    if [ ! -f "$file" ]; then
        log ERROR "File '$file' does not exist"
        return 1
    fi

    mkdir -p "$backup_dir"
    local backup_name="$(basename "$file").backup.$(date +%Y%m%d_%H%M%S)"

    if cp "$file" "$backup_dir/$backup_name"; then
        log INFO "Backup created: $backup_dir/$backup_name"
        echo "$backup_dir/$backup_name"
        return 0
    else
        log ERROR "Failed to create backup"
        return 1
    fi
}

# Example usage
if [[ "${BASH_SOURCE[0]}" == "${0}" ]]; then
    # Test the functions
    log INFO "Testing utility functions"
```

```
    if command_exists "git"; then
        log INFO "Git is available"
    else
        log WARN "Git is not installed"
    fi

    if validate_email "user@example.com"; then
        log INFO "Email format is valid"
    else
        log ERROR "Invalid email format"
    fi

    echo "File extension of script.sh: $(get_extension "script.sh")"
    echo "1048576 bytes = $(bytes_to_human 1048576)"
fi
```

Configuration Manager

```
#!/bin/bash
# config_manager.sh - Configuration file manager

CONFIG_FILE="app.conf"

# Default configuration
declare -A config=(
    [app_name]="MyApp"
    [version]="1.0.0"
    [debug]="false"
    [port]="8080"
    [database_host]="localhost"
    [database_port]="5432"
    [log_level]="INFO"
)

# Load configuration from file
load_config() {
    if [ -f "$CONFIG_FILE" ]; then
        while IFS='=' read -r key value; do
            # Skip comments and empty lines
            [[ $key =~ ^[[:space:]]*# ]] && continue
            [[ -z $key ]] && continue

            # Remove leading/trailing whitespace
            key=$(echo "$key" | xargs)
            value=$(echo "$value" | xargs)
```

```
            config["$key"]="$value"
        done < "$CONFIG_FILE"
        echo "Configuration loaded from $CONFIG_FILE"
    else
        echo "Configuration file not found, using defaults"
    fi
}

# Save configuration to file
save_config() {
    echo "# Application Configuration" > "$CONFIG_FILE"
    echo "# Generated on $(date)" >> "$CONFIG_FILE"
    echo >> "$CONFIG_FILE"

    for key in "${!config[@]}"; do
        echo "$key=${config[$key]}" >> "$CONFIG_FILE"
    done

    echo "Configuration saved to $CONFIG_FILE"
}

# Get configuration value
get_config() {
    local key="$1"
    echo "${config[$key]}"
}

# Set configuration value
set_config() {
    local key="$1"
    local value="$2"
    config["$key"]="$value"
    echo "Set $key = $value"
}

# Show all configuration
show_config() {
    echo "Current Configuration:"
    echo "======================"
    for key in "${!config[@]}"; do
        printf "%-20s = %s\n" "$key" "${config[$key]}"
    done
}

# Interactive configuration editor
edit_config() {
```

```
    while true; do
        echo
        echo "Configuration Editor"
        echo "===================="
        echo "1. Show current config"
        echo "2. Edit a value"
        echo "3. Save config"
        echo "4. Load config"
        echo "5. Exit"

        read -p "Choose option: " choice

        case $choice in
            1) show_config ;;
            2)
                read -p "Enter key to edit: " key
                if [[ -n "${config[$key]}" ]]; then
                    echo "Current value: ${config[$key]}"
                    read -p "Enter new value: " value
                    set_config "$key" "$value"
                else
                    echo "Key '$key' not found"
                fi
                ;;
            3) save_config ;;
            4) load_config ;;
            5) break ;;
            *) echo "Invalid choice" ;;
        esac
    done
}

# Main execution
load_config

if [ $# -eq 0 ]; then
    edit_config
else
    case "$1" in
        get) get_config "$2" ;;
        set) set_config "$2" "$3" ;;
        show) show_config ;;
        save) save_config ;;
        load) load_config ;;
        *) echo "Usage: $0 {get|set|show|save|load} [key] [value]" ;;
    esac
fi
```

Database Backup Script

```
#!/bin/bash
# db_backup.sh - Database backup script with rotation

# Configuration
DB_HOST="${DB_HOST:-localhost}"
DB_PORT="${DB_PORT:-5432}"
DB_USER="${DB_USER:-postgres}"
DB_NAME="${DB_NAME:-myapp}"
BACKUP_DIR="${BACKUP_DIR:-/backup/database}"
RETENTION_DAYS="${RETENTION_DAYS:-7}"
COMPRESSION="${COMPRESSION:-gzip}"

# Logging function
log() {
    echo "$(date '+%Y-%m-%d %H:%M:%S') - $*" | tee -a "$BACKUP_DIR/backup.log"
}

# Check dependencies
check_dependencies() {
    local missing_deps=()

    if ! command -v pg_dump >/dev/null 2>&1; then
        missing_deps+=("postgresql-client")
    fi

    if [ "$COMPRESSION" = "gzip" ] && ! command -v gzip >/dev/null 2>&1; then
        missing_deps+=("gzip")
    fi

    if [ ${#missing_deps[@]} -gt 0 ]; then
        log "ERROR: Missing dependencies: ${missing_deps[*]}"
        exit 1
    fi
}

# Create backup directory
setup_backup_dir() {
    if [ ! -d "$BACKUP_DIR" ]; then
        mkdir -p "$BACKUP_DIR"
        log "INFO: Created backup directory: $BACKUP_DIR"
    fi
}

# Perform database backup
```

```
backup_database() {
    local timestamp=$(date +%Y%m%d_%H%M%S)
    local backup_file="$BACKUP_DIR/${DB_NAME}_${timestamp}.sql"

    log "INFO: Starting backup of database '$DB_NAME'"

    # Set password if provided
    if [ -n "$DB_PASSWORD" ]; then
        export PGPASSWORD="$DB_PASSWORD"
    fi

    # Perform backup
    if pg_dump -h "$DB_HOST" -p "$DB_PORT" -U "$DB_USER" -d "$DB_NAME" > "$backup_file"; the
        log "INFO: Database backup completed: $backup_file"

        # Compress if requested
        if [ "$COMPRESSION" = "gzip" ]; then
            if gzip "$backup_file"; then
                backup_file="${backup_file}.gz"
                log "INFO: Backup compressed: $backup_file"
            else
                log "WARN: Compression failed"
            fi
        fi

        # Calculate file size
        local file_size=$(stat -c%s "$backup_file" 2>/dev/null || stat -f%z "$backup_file" 2
        log "INFO: Backup size: $(numfmt --to=iec-i --suffix=B $file_size)"

        echo "$backup_file"
    else
        log "ERROR: Database backup failed"
        return 1
    fi
}

# Clean old backups
cleanup_old_backups() {
    log "INFO: Cleaning up backups older than $RETENTION_DAYS days"

    local deleted_count=0
    while IFS= read -r -d '' file; do
        rm "$file"
        log "INFO: Deleted old backup: $(basename "$file")"
        ((deleted_count++))
    done < <(find "$BACKUP_DIR" -name "${DB_NAME}_*.sql*" -mtime +$RETENTION_DAYS -print0)
```

```
    log "INFO: Deleted $deleted_count old backup(s)"
}

# Verify backup integrity
verify_backup() {
    local backup_file="$1"

    if [ ! -f "$backup_file" ]; then
        log "ERROR: Backup file not found: $backup_file"
        return 1
    fi

    # Check if file is compressed
    if [[ "$backup_file" == *.gz ]]; then
        if gzip -t "$backup_file"; then
            log "INFO: Backup file integrity verified (compressed)"
            return 0
        else
            log "ERROR: Backup file is corrupted (compressed)"
            return 1
        fi
    else
        # Check SQL file for basic structure
        if grep -q "PostgreSQL database dump" "$backup_file"; then
            log "INFO: Backup file integrity verified"
            return 0
        else
            log "ERROR: Backup file appears to be corrupted"
            return 1
        fi
    fi
}

# Send notification (email or webhook)
send_notification() {
    local status="$1"
    local message="$2"

    if [ -n "$NOTIFICATION_EMAIL" ] && command -v mail >/dev/null 2>&1; then
        echo "$message" | mail -s "Database Backup $status" "$NOTIFICATION_EMAIL"
        log "INFO: Notification sent to $NOTIFICATION_EMAIL"
    fi

    if [ -n "$WEBHOOK_URL" ] && command -v curl >/dev/null 2>&1; then
        curl -X POST -H "Content-Type: application/json" \
             -d "{\"text\":\"Database Backup $status: $message\"}" \
             "$WEBHOOK_URL" >/dev/null 2>&1
```

```
        log "INFO: Webhook notification sent"
    fi
}

# Main backup process
main() {
    log "INFO: Starting database backup process"

    check_dependencies
    setup_backup_dir

    if backup_file=$(backup_database); then
        if verify_backup "$backup_file"; then
            cleanup_old_backups
            send_notification "SUCCESS" "Database backup completed successfully: $(basename
            log "INFO: Backup process completed successfully"
        else
            send_notification "FAILED" "Backup verification failed"
            log "ERROR: Backup verification failed"
            exit 1
        fi
    else
        send_notification "FAILED" "Database backup failed"
        log "ERROR: Backup process failed"
        exit 1
    fi
}

# Handle command line arguments
case "${1:-backup}" in
    backup) main ;;
    verify) verify_backup "$2" ;;
    cleanup) cleanup_old_backups ;;
    *)
        echo "Usage: $0 {backup|verify|cleanup}"
        echo "Environment variables:"
        echo "  DB_HOST, DB_PORT, DB_USER, DB_PASSWORD, DB_NAME"
        echo "  BACKUP_DIR, RETENTION_DAYS, COMPRESSION"
        echo "  NOTIFICATION_EMAIL, WEBHOOK_URL"
        exit 1
        ;;
esac
```

File Processing Examples

Log Analyzer

```
#!/bin/bash
# log_analyzer.sh - Analyze web server logs

LOG_FILE="${1:-/var/log/apache2/access.log}"
OUTPUT_DIR="${2:-./log_analysis}"

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

# Create output directory
mkdir -p "$OUTPUT_DIR"

echo "Analyzing log file: $LOG_FILE"
echo "Output directory: $OUTPUT_DIR"
echo "================================="

# Basic statistics
echo "Generating basic statistics..."
total_requests=$(wc -l < "$LOG_FILE")
echo "Total requests: $total_requests" > "$OUTPUT_DIR/summary.txt"

# Unique IP addresses
unique_ips=$(awk '{print $1}' "$LOG_FILE" | sort -u | wc -l)
echo "Unique IP addresses: $unique_ips" >> "$OUTPUT_DIR/summary.txt"

# Date range
first_date=$(head -1 "$LOG_FILE" | awk '{print $4}' | tr -d '[')
last_date=$(tail -1 "$LOG_FILE" | awk '{print $4}' | tr -d '[')
echo "Date range: $first_date to $last_date" >> "$OUTPUT_DIR/summary.txt"

# Top 10 IP addresses
echo "Generating top IP addresses..."
awk '{print $1}' "$LOG_FILE" | sort | uniq -c | sort -nr | head -10 > "$OUTPUT_DIR/top_ips.t

# Top 10 requested pages
echo "Generating top requested pages..."
awk '{print $7}' "$LOG_FILE" | sort | uniq -c | sort -nr | head -10 > "$OUTPUT_DIR/top_pages

# HTTP status codes
```

```
echo "Analyzing HTTP status codes..."
awk '{print $9}' "$LOG_FILE" | sort | uniq -c | sort -nr > "$OUTPUT_DIR/status_codes.txt"

# User agents
echo "Analyzing user agents..."
awk -F'"' '{print $6}' "$LOG_FILE" | sort | uniq -c | sort -nr | head -20 > "$OUTPUT_DIR/use

# Hourly traffic distribution
echo "Generating hourly traffic distribution..."
awk '{print $4}' "$LOG_FILE" | cut -d: -f2 | sort | uniq -c > "$OUTPUT_DIR/hourly_traffic.tx

# Error analysis (4xx and 5xx)
echo "Analyzing errors..."
awk '$9 ~ /^[45]/ {print $9, $7}' "$LOG_FILE" | sort | uniq -c | sort -nr > "$OUTPUT_DIR/err

# Bandwidth usage (if log format includes bytes)
echo "Calculating bandwidth usage..."
if awk '{print $10}' "$LOG_FILE" | head -1 | grep -q '^[0-9]*$'; then
    total_bytes=$(awk '{sum += $10} END {print sum}' "$LOG_FILE")
    echo "Total bandwidth: $(numfmt --to=iec-i --suffix=B $total_bytes)" >> "$OUTPUT_DIR/sum
fi

# Generate HTML report
echo "Generating HTML report..."
cat > "$OUTPUT_DIR/report.html" << EOF
<!DOCTYPE html>
<html>
<head>
    <title>Log Analysis Report</title>
    <style>
        body { font-family: Arial, sans-serif; margin: 20px; }
        .section { margin: 20px 0; }
        .data { background: #f5f5f5; padding: 10px; border-radius: 5px; }
        pre { white-space: pre-wrap; }
    </style>
</head>
<body>
    <h1>Log Analysis Report</h1>
    <p>Generated on: $(date)</p>

    <div class="section">
        <h2>Summary</h2>
        <div class="data">
            <pre>$(cat "$OUTPUT_DIR/summary.txt")</pre>
        </div>
    </div>
```

```
    <div class="section">
        <h2>Top 10 IP Addresses</h2>
        <div class="data">
            <pre>$(cat "$OUTPUT_DIR/top_ips.txt")</pre>
        </div>
    </div>

    <div class="section">
        <h2>Top 10 Requested Pages</h2>
        <div class="data">
            <pre>$(cat "$OUTPUT_DIR/top_pages.txt")</pre>
        </div>
    </div>

    <div class="section">
        <h2>HTTP Status Codes</h2>
        <div class="data">
            <pre>$(cat "$OUTPUT_DIR/status_codes.txt")</pre>
        </div>
    </div>
</body>
</html>
EOF

echo "Analysis complete! Results saved in: $OUTPUT_DIR"
echo "View the HTML report: $OUTPUT_DIR/report.html"
```

File Organizer

```
#!/bin/bash
# file_organizer.sh - Organize files by type and date

SOURCE_DIR="${1:-.}"
TARGET_DIR="${2:-./organized}"
DRY_RUN="${DRY_RUN:-false}"

# File type associations
declare -A file_types=(
    ["jpg,jpeg,png,gif,bmp,tiff"]="Images"
    ["mp4,avi,mkv,mov,wmv,flv"]="Videos"
    ["mp3,wav,flac,aac,ogg"]="Audio"
    ["pdf,doc,docx,txt,rtf,odt"]="Documents"
    ["zip,rar,7z,tar,gz,bz2"]="Archives"
    ["exe,msi,deb,rpm,dmg"]="Executables"
    ["html,css,js,php,py,java,cpp,c"]="Code"
```

```
)

# Logging function
log() {
    echo "$(date '+%H:%M:%S') - $*"
}

# Get file category based on extension
get_file_category() {
    local filename="$1"
    local extension="${filename##*.}"
    extension=$(echo "$extension" | tr '[:upper:]' '[:lower:]')

    for types in "${!file_types[@]}"; do
        if [[ ",$types," == *",$extension,"* ]]; then
            echo "${file_types[$types]}"
            return
        fi
    done

    echo "Other"
}

# Create directory structure
create_directory() {
    local dir="$1"
    if [ "$DRY_RUN" = "true" ]; then
        log "DRY RUN: Would create directory: $dir"
    else
        mkdir -p "$dir"
        log "Created directory: $dir"
    fi
}

# Move file
move_file() {
    local source="$1"
    local target="$2"

    if [ "$DRY_RUN" = "true" ]; then
        log "DRY RUN: Would move '$source' to '$target'"
    else
        if mv "$source" "$target"; then
            log "Moved: $(basename "$source") -> $target"
        else
            log "ERROR: Failed to move $(basename "$source")"
        fi
```

```
    fi
}

# Organize files
organize_files() {
    local source_dir="$1"
    local target_dir="$2"
    local file_count=0
    local dir_count=0

    log "Starting file organization..."
    log "Source: $source_dir"
    log "Target: $target_dir"

    # Process all files in source directory
    find "$source_dir" -maxdepth 1 -type f | while read -r file; do
        # Skip hidden files unless specified
        if [[ "$(basename "$file")" == .* ]] && [ "$INCLUDE_HIDDEN" != "true" ]; then
            continue
        fi

        # Get file info
        filename=$(basename "$file")
        category=$(get_file_category "$filename")

        # Get file date
        if command -v stat >/dev/null 2>&1; then
            file_date=$(stat -c %Y "$file" 2>/dev/null || stat -f %m "$file" 2>/dev/null)
            year=$(date -d "@$file_date" +%Y 2>/dev/null || date -r "$file_date" +%Y 2>/dev/
            month=$(date -d "@$file_date" +%m 2>/dev/null || date -r "$file_date" +%m 2>/dev
        else
            year=$(date +%Y)
            month=$(date +%m)
        fi

        # Create target directory structure
        target_path="$target_dir/$category/$year/$month"
        create_directory "$target_path"

        # Move file
        move_file "$file" "$target_path/$filename"
        ((file_count++))
    done

    log "Organization complete. Processed $file_count files."
}
```

```
# Generate report
generate_report() {
    local target_dir="$1"
    local report_file="$target_dir/organization_report.txt"

    log "Generating organization report..."

    {
        echo "File Organization Report"
        echo "========================"
        echo "Generated on: $(date)"
        echo "Target directory: $target_dir"
        echo

        echo "Directory Structure:"
        echo "===================="
        if [ -d "$target_dir" ]; then
            find "$target_dir" -type d | sort
        fi
        echo

        echo "File Count by Category:"
        echo "======================="
        for category in "${file_types[@]}" "Other"; do
            if [ -d "$target_dir/$category" ]; then
                count=$(find "$target_dir/$category" -type f | wc -l)
                printf "%-15s: %d files\n" "$category" "$count"
            fi
        done

    } > "$report_file"

    log "Report saved: $report_file"
}

# Show usage
show_usage() {
    cat << EOF
Usage: $0 [source_dir] [target_dir]

Options:
  source_dir    Source directory to organize (default: current directory)
  target_dir    Target directory for organized files (default: ./organized)

Environment Variables:
  DRY_RUN=true         Show what would be done without actually moving files
  INCLUDE_HIDDEN=true  Include hidden files in organization
```

```
Examples:
  $0                          # Organize current directory
  $0 /home/user/Downloads     # Organize Downloads folder
  DRY_RUN=true $0             # Preview organization without moving files

File Categories:
EOF

    for types in "${!file_types[@]}"; do
        printf "  %-15s: %s\n" "${file_types[$types]}" "$types"
    done
}

# Main execution
main() {
    if [ "$1" = "-h" ] || [ "$1" = "--help" ]; then
        show_usage
        exit 0
    fi

    if [ ! -d "$SOURCE_DIR" ]; then
        log "ERROR: Source directory '$SOURCE_DIR' does not exist"
        exit 1
    fi

    if [ "$DRY_RUN" = "true" ]; then
        log "DRY RUN MODE - No files will be moved"
    fi

    organize_files "$SOURCE_DIR" "$TARGET_DIR"

    if [ "$DRY_RUN" != "true" ]; then
        generate_report "$TARGET_DIR"
    fi
}

main "$@"
```

CSV Processor

```
#!/bin/bash
# csv_processor.sh - Process and analyze CSV files

CSV_FILE="$1"
DELIMITER="${DELIMITER:-,}"
```

```
OUTPUT_FORMAT="${OUTPUT_FORMAT:-table}"

# Check if file exists
if [ ! -f "$CSV_FILE" ]; then
    echo "Usage: $0 <csv_file>"
    echo "Environment variables:"
    echo "  DELIMITER     - Field delimiter (default: ,)"
    echo "  OUTPUT_FORMAT - Output format: table, json, html (default: table)"
    exit 1
fi

# Function to get CSV headers
get_headers() {
    head -1 "$CSV_FILE" | tr "$DELIMITER" '\n' | nl -w2 -s': '
}

# Function to count rows
count_rows() {
    local total_rows=$(wc -l < "$CSV_FILE")
    local data_rows=$((total_rows - 1))
    echo "Total rows: $total_rows (including header)"
    echo "Data rows: $data_rows"
}

# Function to show column statistics
column_stats() {
    local column_num="$1"
    local column_name=$(head -1 "$CSV_FILE" | cut -d"$DELIMITER" -f"$column_num")

    echo "Statistics for column $column_num: $column_name"
    echo "=============================================="

    # Get column data (skip header)
    tail -n +2 "$CSV_FILE" | cut -d"$DELIMITER" -f"$column_num" > /tmp/column_data.tmp

    # Count non-empty values
    local non_empty=$(grep -v '^$' /tmp/column_data.tmp | wc -l)
    local empty=$(grep '^$' /tmp/column_data.tmp | wc -l)

    echo "Non-empty values: $non_empty"
    echo "Empty values: $empty"

    # Check if column contains numbers
    if grep -q '^[0-9]*\.?[0-9]*$' /tmp/column_data.tmp; then
        echo "Data type: Numeric"

        # Calculate numeric statistics
```

```
        local sum=$(awk '{sum += $1} END {print sum}' /tmp/column_data.tmp)
        local count=$(grep -v '^$' /tmp/column_data.tmp | wc -l)
        local avg=$(echo "scale=2; $sum / $count" | bc -l 2>/dev/null || echo "N/A")
        local min=$(sort -n /tmp/column_data.tmp | head -1)
        local max=$(sort -n /tmp/column_data.tmp | tail -1)

        echo "Sum: $sum"
        echo "Average: $avg"
        echo "Min: $min"
        echo "Max: $max"
    else
        echo "Data type: Text"

        # Show unique values count
        local unique_count=$(sort /tmp/column_data.tmp | uniq | wc -l)
        echo "Unique values: $unique_count"

        # Show most common values
        echo "Most common values:"
        sort /tmp/column_data.tmp | uniq -c | sort -nr | head -5
    fi

    rm -f /tmp/column_data.tmp
}

# Function to filter CSV data
filter_data() {
    local column="$1"
    local operator="$2"
    local value="$3"

    echo "Filtering: Column $column $operator '$value'"
    echo "=========================================="

    # Show header
    head -1 "$CSV_FILE"

    # Apply filter based on operator
    case "$operator" in
        "="|"eq")
            tail -n +2 "$CSV_FILE" | awk -F"$DELIMITER" -v col="$column" -v val="$value" '$c
            ;;
        "!="|"ne")
            tail -n +2 "$CSV_FILE" | awk -F"$DELIMITER" -v col="$column" -v val="$value" '$c
            ;;
        ">"|"gt")
            tail -n +2 "$CSV_FILE" | awk -F"$DELIMITER" -v col="$column" -v val="$value" '$c
```

```
            ;;
        "<"|"lt")
            tail -n +2 "$CSV_FILE" | awk -F"$DELIMITER" -v col="$column" -v val="$value" '$c
            ;;
        "contains")
            tail -n +2 "$CSV_FILE" | awk -F"$DELIMITER" -v col="$column" -v val="$value" 'in
            ;;
        *)
            echo "Supported operators: =, !=, >, <, contains"
            return 1
            ;;
    esac
}

# Function to convert to JSON
to_json() {
    local headers=($(head -1 "$CSV_FILE" | tr "$DELIMITER" ' '))

    echo "["
    local first_row=true

    tail -n +2 "$CSV_FILE" | while IFS="$DELIMITER" read -r -a fields; do
        if [ "$first_row" = true ]; then
            first_row=false
        else
            echo -n ","
        fi

        echo -n "  {"
        for i in "${!headers[@]}"; do
            if [ $i -gt 0 ]; then
                echo -n ", "
            fi
            echo -n "\"${headers[i]}\": \"${fields[i]}\""
        done
        echo -n "}"
    done

    echo
    echo "]"
}

# Function to convert to HTML table
to_html() {
    echo "<table border='1'>"
    echo "<thead><tr>"
```

```
    # Headers
    head -1 "$CSV_FILE" | tr "$DELIMITER" '\n' | while read -r header; do
        echo "<th>$header</th>"
    done

    echo "</tr></thead><tbody>"

    # Data rows
    tail -n +2 "$CSV_FILE" | while IFS="$DELIMITER" read -r -a fields; do
        echo "<tr>"
        for field in "${fields[@]}"; do
            echo "<td>$field</td>"
        done
        echo "</tr>"
    done

    echo "</tbody></table>"
}

# Function to show menu
show_menu() {
    echo
    echo "CSV Processor Menu"
    echo "=================="
    echo "1. Show headers"
    echo "2. Count rows"
    echo "3. Column statistics"
    echo "4. Filter data"
    echo "5. Convert to JSON"
    echo "6. Convert to HTML"
    echo "7. Show first 10 rows"
    echo "8. Show last 10 rows"
    echo "9. Exit"
    echo
}

# Interactive mode
interactive_mode() {
    while true; do
        show_menu
        read -p "Choose option: " choice

        case $choice in
            1)
                echo "CSV Headers:"
                get_headers
                ;;
```

```
            2)
                count_rows
                ;;
            3)
                echo "Available columns:"
                get_headers
                read -p "Enter column number: " col_num
                column_stats "$col_num"
                ;;
            4)
                echo "Available columns:"
                get_headers
                read -p "Enter column number: " col_num
                read -p "Enter operator (=, !=, >, <, contains): " operator
                read -p "Enter value: " value
                filter_data "$col_num" "$operator" "$value"
                ;;
            5)
                to_json
                ;;
            6)
                to_html
                ;;
            7)
                echo "First 10 rows:"
                head -11 "$CSV_FILE" | column -t -s"$DELIMITER"
                ;;
            8)
                echo "Last 10 rows:"
                tail -10 "$CSV_FILE" | column -t -s"$DELIMITER"
                ;;
            9)
                echo "Goodbye!"
                exit 0
                ;;
            *)
                echo "Invalid choice"
                ;;
        esac

        echo
        read -p "Press Enter to continue..."
    done
}

# Main execution
```

```
echo "Processing CSV file: $CSV_FILE"
echo "Delimiter: '$DELIMITER'"
echo

# If no additional arguments, start interactive mode
if [ $# -eq 1 ]; then
    interactive_mode
else
    # Command line mode
    case "$2" in
        headers) get_headers ;;
        count) count_rows ;;
        stats) column_stats "$3" ;;
        filter) filter_data "$3" "$4" "$5" ;;
        json) to_json ;;
        html) to_html ;;
        *)
            echo "Usage: $0 <csv_file> [command] [args...]"
            echo "Commands: headers, count, stats <col>, filter <col> <op> <val>, json, html
            ;;
    esac
fi
```

System Administration Examples

System Monitor

```
#!/bin/bash
# system_monitor.sh - Comprehensive system monitoring script

ALERT_CPU_THRESHOLD=${ALERT_CPU_THRESHOLD:-80}
ALERT_MEMORY_THRESHOLD=${ALERT_MEMORY_THRESHOLD:-85}
ALERT_DISK_THRESHOLD=${ALERT_DISK_THRESHOLD:-90}
LOG_FILE=${LOG_FILE:-"/var/log/system_monitor.log"}
EMAIL_ALERT=${EMAIL_ALERT:-""}
WEBHOOK_URL=${WEBHOOK_URL:-""}

# Colors for output
RED='\033[0;31m'
YELLOW='\033[1;33m'
GREEN='\033[0;32m'
NC='\033[0m' # No Color

# Logging function
```

```
log_message() {
    local level="$1"
    shift
    local message="$*"
    local timestamp=$(date '+%Y-%m-%d %H:%M:%S')

    echo "[$timestamp] [$level] $message" >> "$LOG_FILE"

    case "$level" in
        ERROR) echo -e "${RED}[$timestamp] ERROR: $message${NC}" ;;
        WARN)  echo -e "${YELLOW}[$timestamp] WARN:  $message${NC}" ;;
        INFO)  echo -e "${GREEN}[$timestamp] INFO:  $message${NC}" ;;
    esac
}

# Send alert notification
send_alert() {
    local subject="$1"
    local message="$2"

    # Email notification
    if [ -n "$EMAIL_ALERT" ] && command -v mail >/dev/null 2>&1; then
        echo "$message" | mail -s "$subject" "$EMAIL_ALERT"
        log_message INFO "Alert sent via email to $EMAIL_ALERT"
    fi

    # Webhook notification
    if [ -n "$WEBHOOK_URL" ] && command -v curl >/dev/null 2>&1; then
        curl -X POST -H "Content-Type: application/json" \
             -d "{\"text\":\"$subject: $message\"}" \
             "$WEBHOOK_URL" >/dev/null 2>&1
        log_message INFO "Alert sent via webhook"
    fi
}

# Check CPU usage
check_cpu() {
    local cpu_usage=$(top -bn1 | grep "Cpu(s)" | awk '{print $2}' | cut -d'%' -f1)
    cpu_usage=${cpu_usage%.*}  # Remove decimal part

    echo "CPU Usage: ${cpu_usage}%"

    if [ "$cpu_usage" -gt "$ALERT_CPU_THRESHOLD" ]; then
        local message="High CPU usage detected: ${cpu_usage}% (threshold: ${ALERT_CPU_THRESH
        log_message WARN "$message"
        send_alert "CPU Alert" "$message"
```

```
        # Show top processes
        echo "Top CPU consuming processes:"
        ps aux --sort=-%cpu | head -6
    else
        log_message INFO "CPU usage normal: ${cpu_usage}%"
    fi
}

# Check memory usage
check_memory() {
    local memory_info=$(free | grep Mem)
    local total=$(echo $memory_info | awk '{print $2}')
    local used=$(echo $memory_info | awk '{print $3}')
    local memory_usage=$((used * 100 / total))

    echo "Memory Usage: ${memory_usage}% ($(numfmt --to=iec $((used * 1024))) / $(numfmt --t

    if [ "$memory_usage" -gt "$ALERT_MEMORY_THRESHOLD" ]; then
        local message="High memory usage detected: ${memory_usage}% (threshold: ${ALERT_MEMO
        log_message WARN "$message"
        send_alert "Memory Alert" "$message"

        # Show top memory consuming processes
        echo "Top memory consuming processes:"
        ps aux --sort=-%mem | head -6
    else
        log_message INFO "Memory usage normal: ${memory_usage}%"
    fi
}

# Check disk usage
check_disk() {
    echo "Disk Usage:"
    df -h | grep -vE '^Filesystem|tmpfs|cdrom' | while read output; do
        usage=$(echo $output | awk '{print $5}' | cut -d'%' -f1)
        partition=$(echo $output | awk '{print $1}')
        mount_point=$(echo $output | awk '{print $6}')

        echo "  $mount_point: ${usage}%"

        if [ "$usage" -gt "$ALERT_DISK_THRESHOLD" ]; then
            local message="High disk usage detected on $mount_point: ${usage}% (threshold: $
            log_message WARN "$message"
            send_alert "Disk Alert" "$message"
        else
            log_message INFO "Disk usage normal on $mount_point: ${usage}%"
        fi
```

```
    done
}

# Check system load
check_load() {
    local load_avg=$(uptime | awk -F'load average:' '{print $2}')
    local load_1min=$(echo $load_avg | awk '{print $1}' | tr -d ',')
    local cpu_cores=$(nproc)

    echo "Load Average: $load_avg"
    echo "CPU Cores: $cpu_cores"

    # Check if 1-minute load is higher than number of cores
    if (( $(echo "$load_1min > $cpu_cores" | bc -l) )); then
        local message="High system load detected: $load_1min (cores: $cpu_cores)"
        log_message WARN "$message"
        send_alert "Load Alert" "$message"
    else
        log_message INFO "System load normal: $load_1min"
    fi
}

# Check running services
check_services() {
    local services=("ssh" "nginx" "apache2" "mysql" "postgresql")

    echo "Service Status:"
    for service in "${services[@]}"; do
        if systemctl is-active --quiet "$service" 2>/dev/null; then
            echo "  $service: Running"
            log_message INFO "Service $service is running"
        elif systemctl list-unit-files | grep -q "^$service.service"; then
            echo "  $service: Stopped"
            log_message WARN "Service $service is stopped"
            send_alert "Service Alert" "Service $service is not running"
        fi
    done
}

# Check network connectivity
check_network() {
    local test_hosts=("8.8.8.8" "1.1.1.1" "google.com")

    echo "Network Connectivity:"
    for host in "${test_hosts[@]}"; do
        if ping -c 1 -W 5 "$host" >/dev/null 2>&1; then
            echo "  $host: OK"
```

```
            log_message INFO "Network connectivity to $host OK"
        else
            echo "  $host: FAILED"
            log_message ERROR "Network connectivity to $host failed"
            send_alert "Network Alert" "Cannot reach $host"
        fi
    done
}

# Check log files for errors
check_logs() {
    local log_files=("/var/log/syslog" "/var/log/auth.log" "/var/log/kern.log")
    local time_threshold="1 hour ago"

    echo "Recent Log Errors:"
    for log_file in "${log_files[@]}"; do
        if [ -f "$log_file" ]; then
            local error_count=$(find "$log_file" -newermt "$time_threshold" -exec grep -i "e
            echo "  $(basename "$log_file"): $error_count errors in last hour"

            if [ "$error_count" -gt 10 ]; then
                log_message WARN "High error count in $log_file: $error_count"
                send_alert "Log Alert" "High error count in $log_file: $error_count errors i
            fi
        fi
    done
}

# Generate system report
generate_report() {
    local report_file="/tmp/system_report_$(date +%Y%m%d_%H%M%S).txt"

    {
        echo "System Monitoring Report"
        echo "========================"
        echo "Generated: $(date)"
        echo "Hostname: $(hostname)"
        echo

        echo "System Information:"
        echo "==================="
        uname -a
        echo

        echo "Uptime:"
        echo "======="
        uptime
```

```
        echo

        echo "CPU Information:"
        echo "================"
        lscpu | grep -E "Model name|CPU\(s\)|Thread|Core"
        echo

        echo "Memory Information:"
        echo "==================="
        free -h
        echo

        echo "Disk Usage:"
        echo "============"
        df -h
        echo

        echo "Network Interfaces:"
        echo "==================="
        ip addr show
        echo

        echo "Running Processes (Top 10 by CPU):"
        echo "==================================="
        ps aux --sort=-%cpu | head -11
        echo

        echo "Running Processes (Top 10 by Memory):"
        echo "======================================"
        ps aux --sort=-%mem | head -11

    } > "$report_file"

    echo "System report generated: $report_file"
    log_message INFO "System report generated: $report_file"
}

# Main monitoring function
run_monitoring() {
    log_message INFO "Starting system monitoring"

    echo "System Monitoring Report - $(date)"
    echo "=================================="
    echo

    check_cpu
    echo
```

```
    check_memory
    echo

    check_disk
    echo

    check_load
    echo

    check_services
    echo

    check_network
    echo

    check_logs
    echo

    log_message INFO "System monitoring completed"
}

# Show usage
show_usage() {
    cat << EOF
Usage: $0 [options]

Options:
  -c, --continuous    Run monitoring continuously
  -i, --interval N    Set monitoring interval in seconds (default: 300)
  -r, --report        Generate detailed system report
  -h, --help          Show this help message

Environment Variables:
  ALERT_CPU_THRESHOLD     CPU usage alert threshold (default: 80)
  ALERT_MEMORY_THRESHOLD  Memory usage alert threshold (default: 85)
  ALERT_DISK_THRESHOLD    Disk usage alert threshold (default: 90)
  LOG_FILE               Log file path (default: /var/log/system_monitor.log)
  EMAIL_ALERT            Email address for alerts
  WEBHOOK_URL            Webhook URL for alerts

Examples:
  $0                      # Run monitoring once
  $0 -c                   # Run continuous monitoring
  $0 -c -i 60            # Run continuous monitoring every 60 seconds
  $0 -r                   # Generate detailed report
EOF
```

```
}

# Parse command line arguments
CONTINUOUS=false
INTERVAL=300

while [[ $# -gt 0 ]]; do
    case $1 in
        -c|--continuous)
            CONTINUOUS=true
            shift
            ;;
        -i|--interval)
            INTERVAL="$2"
            shift 2
            ;;
        -r|--report)
            generate_report
            exit 0
            ;;
        -h|--help)
            show_usage
            exit 0
            ;;
        *)
            echo "Unknown option: $1"
            show_usage
            exit 1
            ;;
    esac
done

# Main execution
if [ "$CONTINUOUS" = true ]; then
    log_message INFO "Starting continuous monitoring (interval: ${INTERVAL}s)"
    while true; do
        run_monitoring
        echo "Sleeping for $INTERVAL seconds..."
        sleep "$INTERVAL"
    done
else
    run_monitoring
fi
```

This workbook provides a comprehensive collection of Bash scripting examples ranging from basic to intermediate level. Each example includes practical, real-world scenarios that demonstrate different aspects of Bash programming. The examples cover:

1. **Basic Examples**: Variables, user input, calculations, file operations
2. **Intermediate Examples**: Functions, configuration management, database operations
3. **File Processing**: Log analysis, file organization, CSV processing
4. **System Administration**: Comprehensive system monitoring

These examples can be used as learning materials, reference implementations, or starting points for your own scripts.

Advanced Bash Examples

This section contains advanced Bash scripting examples that demonstrate complex concepts, patterns, and real-world applications.

Process Management and IPC

Process Pool Manager

```
#!/bin/bash
# process_pool.sh - Manage a pool of worker processes

MAX_WORKERS=${MAX_WORKERS:-4}
WORK_QUEUE="/tmp/work_queue.$$"
WORKER_PIDS=()
RUNNING=true

# Cleanup function
cleanup() {
    RUNNING=false
    echo "Shutting down workers..."

    for pid in "${WORKER_PIDS[@]}"; do
        if kill -0 "$pid" 2>/dev/null; then
            kill "$pid"
        fi
    done

    rm -f "$WORK_QUEUE"
    exit 0
}

# Set up signal handlers
trap cleanup SIGINT SIGTERM

# Worker function
worker() {
    local worker_id="$1"
    echo "Worker $worker_id started (PID: $$)"
```

```
    while $RUNNING; do
        if [ -p "$WORK_QUEUE" ]; then
            if read -t 1 task < "$WORK_QUEUE"; then
                echo "Worker $worker_id processing: $task"
                # Simulate work
                sleep $((RANDOM % 5 + 1))
                echo "Worker $worker_id completed: $task"
            fi
        fi
    done

    echo "Worker $worker_id shutting down"
}

# Create work queue (named pipe)
mkfifo "$WORK_QUEUE"

# Start worker processes
for ((i=1; i<=MAX_WORKERS; i++)); do
    worker "$i" &
    WORKER_PIDS+=($!)
done

echo "Started $MAX_WORKERS workers"
echo "Send tasks by writing to: $WORK_QUEUE"
echo "Press Ctrl+C to shutdown"

# Add some sample tasks
for task in "task1" "task2" "task3" "task4" "task5"; do
    echo "$task" > "$WORK_QUEUE" &
done

# Wait for workers
wait
```#
Distributed Task Runner
```bash
#!/bin/bash
# distributed_runner.sh - Run tasks across multiple servers

SERVERS_FILE="${SERVERS_FILE:-servers.txt}"
SSH_KEY="${SSH_KEY:-~/.ssh/id_rsa}"
PARALLEL_JOBS="${PARALLEL_JOBS:-3}"
TIMEOUT="${TIMEOUT:-300}"

# Server list format: user@hostname:port
declare -a SERVERS
```

```
# Load servers from file
load_servers() {
    if [ ! -f "$SERVERS_FILE" ]; then
        echo "Error: Servers file '$SERVERS_FILE' not found"
        exit 1
    fi

    while IFS= read -r server; do
        [[ $server =~ ^[[:space:]]*# ]] && continue
        [[ -z $server ]] && continue
        SERVERS+=("$server")
    done < "$SERVERS_FILE"

    echo "Loaded ${#SERVERS[@]} servers"
}

# Execute command on remote server
execute_remote() {
    local server="$1"
    local command="$2"
    local output_file="$3"

    local user_host="${server%:*}"
    local port="${server##*:}"
    [ "$port" = "$server" ] && port="22"

    echo "Executing on $user_host: $command" >> "$output_file"

    timeout "$TIMEOUT" ssh -i "$SSH_KEY" -p "$port" -o ConnectTimeout=10 \
        -o StrictHostKeyChecking=no "$user_host" "$command" >> "$output_file" 2>&1

    local exit_code=$?
    echo "Exit code: $exit_code" >> "$output_file"
    return $exit_code
}

# Run command on all servers in parallel
run_distributed() {
    local command="$1"
    local output_dir="results_$(date +%Y%m%d_%H%M%S)"

    mkdir -p "$output_dir"
    echo "Running command on ${#SERVERS[@]} servers: $command"
    echo "Output directory: $output_dir"

    local pids=()
```

```
    local server_index=0

    for server in "${SERVERS[@]}"; do
        local output_file="$output_dir/server_${server_index}.log"

        execute_remote "$server" "$command" "$output_file" &
        pids+=($!)

        ((server_index++))

        # Limit parallel jobs
        if [ ${#pids[@]} -ge $PARALLEL_JOBS ]; then
            wait "${pids[0]}"
            pids=("${pids[@]:1}")
        fi
    done

    # Wait for remaining jobs
    for pid in "${pids[@]}"; do
        wait "$pid"
    done

    echo "All tasks completed. Results in: $output_dir"

    # Generate summary
    generate_summary "$output_dir"
}

# Generate execution summary
generate_summary() {
    local output_dir="$1"
    local summary_file="$output_dir/summary.txt"

    {
        echo "Distributed Execution Summary"
        echo "============================="
        echo "Executed at: $(date)"
        echo "Total servers: ${#SERVERS[@]}"
        echo

        local success_count=0
        local failure_count=0

        for log_file in "$output_dir"/server_*.log; do
            local server_num=$(basename "$log_file" .log | cut -d_ -f2)
            local server="${SERVERS[$server_num]}"
```

```
            if grep -q "Exit code: 0" "$log_file"; then
                echo "  $server - SUCCESS"
                ((success_count++))
            else
                echo "  $server - FAILED"
                ((failure_count++))
            fi
        done

        echo
        echo "Results: $success_count successful, $failure_count failed"

    } > "$summary_file"

    cat "$summary_file"
}

# Main execution
if [ $# -eq 0 ]; then
    echo "Usage: $0 <command>"
    echo "Environment variables:"
    echo "  SERVERS_FILE    - File containing server list (default: servers.txt)"
    echo "  SSH_KEY         - SSH private key (default: ~/.ssh/id_rsa)"
    echo "  PARALLEL_JOBS   - Number of parallel jobs (default: 3)"
    echo "  TIMEOUT         - Command timeout in seconds (default: 300)"
    exit 1
fi

load_servers
run_distributed "$*"
```

Network Programming

HTTP Server in Bash

```
#!/bin/bash
# http_server.sh - Simple HTTP server in pure Bash

PORT="${PORT:-8080}"
DOCUMENT_ROOT="${DOCUMENT_ROOT:-./www}"
LOG_FILE="${LOG_FILE:-access.log}"

# MIME types
declare -A MIME_TYPES=(
    ["html"]="text/html"
```

```
    ["css"]="text/css"
    ["js"]="application/javascript"
    ["json"]="application/json"
    ["txt"]="text/plain"
    ["png"]="image/png"
    ["jpg"]="image/jpeg"
    ["gif"]="image/gif"
)

# Log HTTP request
log_request() {
    local client_ip="$1"
    local method="$2"
    local path="$3"
    local status="$4"
    local size="$5"

    echo "$(date '+%d/%b/%Y:%H:%M:%S %z') $client_ip \"$method $path HTTP/1.1\" $status $siz
}

# Get MIME type from file extension
get_mime_type() {
    local file="$1"
    local extension="${file##*.}"
    echo "${MIME_TYPES[$extension]:-application/octet-stream}"
}

# Send HTTP response
send_response() {
    local status="$1"
    local content_type="$2"
    local content="$3"
    local content_length="${#content}"

    echo "HTTP/1.1 $status"
    echo "Content-Type: $content_type"
    echo "Content-Length: $content_length"
    echo "Connection: close"
    echo
    echo -n "$content"
}

# Handle HTTP request
handle_request() {
    local client_ip="$1"

    # Read request line
```

```
    read -r method path protocol

    # Read headers (until empty line)
    while read -r header && [ -n "$header" ]; do
        header=$(echo "$header" | tr -d '\r')
        # Process headers if needed
    done

    # Remove query string from path
    path="${path%%\?*}"

    # Security: prevent directory traversal
    path="${path//..\/}"

    # Default to index.html for directory requests
    if [[ "$path" == */ ]]; then
        path="${path}index.html"
    fi

    local file_path="$DOCUMENT_ROOT$path"

    if [ -f "$file_path" ]; then
        local content_type=$(get_mime_type "$file_path")
        local content=$(cat "$file_path")
        send_response "200 OK" "$content_type" "$content"
        log_request "$client_ip" "$method" "$path" "200" "${#content}"
    else
        local error_content="<html><body><h1>404 Not Found</h1><p>The requested file was not
        send_response "404 Not Found" "text/html" "$error_content"
        log_request "$client_ip" "$method" "$path" "404" "${#error_content}"
    fi
}

# Start server
start_server() {
    echo "Starting HTTP server on port $PORT"
    echo "Document root: $DOCUMENT_ROOT"
    echo "Log file: $LOG_FILE"

    # Create document root if it doesn't exist
    mkdir -p "$DOCUMENT_ROOT"

    # Create a simple index.html if it doesn't exist
    if [ ! -f "$DOCUMENT_ROOT/index.html" ]; then
        cat > "$DOCUMENT_ROOT/index.html" << EOF
<!DOCTYPE html>
<html>
```

```
<head>
    <title>Bash HTTP Server</title>
</head>
<body>
    <h1>Welcome to Bash HTTP Server</h1>
    <p>Server started at: $(date)</p>
    <p>This is a simple HTTP server written in Bash.</p>
</body>
</html>
EOF
    fi

    # Start listening
    while true; do
        # Use netcat to listen for connections
        if command -v nc >/dev/null 2>&1; then
            nc -l -p "$PORT" -e "$0" handle_connection
        elif command -v socat >/dev/null 2>&1; then
            socat TCP-LISTEN:$PORT,reuseaddr,fork EXEC:"$0 handle_connection"
        else
            echo "Error: netcat or socat required"
            exit 1
        fi
    done
}

# Handle individual connection
handle_connection() {
    local client_ip="${SOCAT_PEERADDR:-unknown}"
    handle_request "$client_ip"
}

# Main execution
case "${1:-start}" in
    start)
        start_server
        ;;
    handle_connection)
        handle_connection
        ;;
    *)
        echo "Usage: $0 [start]"
        exit 1
        ;;
esac
```

Network Scanner

```
#!/bin/bash
# network_scanner.sh - Network discovery and port scanning

NETWORK="${1:-192.168.1.0/24}"
TIMEOUT="${TIMEOUT:-1}"
THREADS="${THREADS:-50}"

# Common ports to scan
COMMON_PORTS=(22 23 25 53 80 110 143 443 993 995 3389 5432 3306)

# Parse CIDR notation
parse_network() {
    local network="$1"
    local ip="${network%/*}"
    local prefix="${network#*/}"

    if [ "$prefix" = "$network" ]; then
        # No CIDR notation, assume single host
        echo "$ip"
        return
    fi

    # Convert CIDR to IP range
    local IFS='.'
    local ip_parts=($ip)
    local ip_int=$((${ip_parts[0]} << 24 | ${ip_parts[1]} << 16 | ${ip_parts[2]} << 8 | ${ip

    local mask=$((0xFFFFFFFF << (32 - prefix)))
    local network_int=$((ip_int & mask))
    local broadcast_int=$((network_int | (0xFFFFFFFF >> prefix)))

    for ((i=network_int+1; i<broadcast_int; i++)); do
        printf "%d.%d.%d.%d\n" $((i >> 24 & 0xFF)) $((i >> 16 & 0xFF)) $((i >> 8 & 0xFF)) $(
    done
}

# Ping sweep to discover live hosts
ping_sweep() {
    local network="$1"
    local output_file="/tmp/live_hosts.$$"

    echo "Performing ping sweep on $network..."

    parse_network "$network" | while read -r ip; do
        {
```

```
            if ping -c 1 -W "$TIMEOUT" "$ip" >/dev/null 2>&1; then
                echo "$ip" >> "$output_file"
                echo "Host alive: $ip"
            fi
        } &

        # Limit concurrent processes
        (($(jobs -r | wc -l) >= THREADS)) && wait
    done

    wait

    if [ -f "$output_file" ]; then
        sort -V "$output_file"
        rm "$output_file"
    fi
}

# Port scan a single host
port_scan() {
    local host="$1"
    local ports=("${@:2}")

    echo "Scanning ports on $host..."

    for port in "${ports[@]}"; do
        {
            if timeout "$TIMEOUT" bash -c "echo >/dev/tcp/$host/$port" 2>/dev/null; then
                echo "$host:$port - OPEN"

                # Try to identify service
                local service=$(getent services "$port/tcp" 2>/dev/null | awk '{print $1}')
                [ -n "$service" ] && echo "  Service: $service"
            fi
        } &

        # Limit concurrent processes
        (($(jobs -r | wc -l) >= THREADS)) && wait
    done

    wait
}

# Comprehensive network scan
full_scan() {
    local network="$1"
    local output_dir="scan_results_$(date +%Y%m%d_%H%M%S)"
```

```
    mkdir -p "$output_dir"

    echo "Starting comprehensive network scan..."
    echo "Network: $network"
    echo "Output directory: $output_dir"

    # Discover live hosts
    echo "Phase 1: Host discovery"
    local live_hosts=()
    while IFS= read -r host; do
        live_hosts+=("$host")
    done < <(ping_sweep "$network")

    echo "Found ${#live_hosts[@]} live hosts"
    echo "${live_hosts[@]}" > "$output_dir/live_hosts.txt"

    # Port scan each live host
    echo "Phase 2: Port scanning"
    for host in "${live_hosts[@]}"; do
        echo "Scanning $host..."
        port_scan "$host" "${COMMON_PORTS[@]}" > "$output_dir/${host}_ports.txt"
    done

    # Generate summary report
    {
        echo "Network Scan Report"
        echo "==================="
        echo "Scan date: $(date)"
        echo "Network: $network"
        echo "Live hosts: ${#live_hosts[@]}"
        echo

        echo "Host Summary:"
        echo "============"
        for host in "${live_hosts[@]}"; do
            local open_ports=$(grep -c "OPEN" "$output_dir/${host}_ports.txt" 2>/dev/null ||
            echo "$host - $open_ports open ports"
        done

        echo
        echo "Open Ports by Service:"
        echo "====================="
        for port in "${COMMON_PORTS[@]}"; do
            local hosts_with_port=()
            for host in "${live_hosts[@]}"; do
                if grep -q "$host:$port - OPEN" "$output_dir/${host}_ports.txt" 2>/dev/null;
```

```
                    hosts_with_port+=("$host")
                fi
            done

            if [ ${#hosts_with_port[@]} -gt 0 ]; then
                local service=$(getent services "$port/tcp" 2>/dev/null | awk '{print $1}' |
                echo "Port $port ($service): ${hosts_with_port[*]}"
            fi
        done

    } > "$output_dir/summary.txt"

    echo "Scan completed. Results in: $output_dir"
    cat "$output_dir/summary.txt"
}

# Main execution
case "${2:-full}" in
    ping)
        ping_sweep "$NETWORK"
        ;;
    port)
        if [ $# -lt 2 ]; then
            echo "Usage: $0 <host> port [port...]"
            exit 1
        fi
        port_scan "$1" "${COMMON_PORTS[@]}"
        ;;
    full)
        full_scan "$NETWORK"
        ;;
    *)
        echo "Usage: $0 <network> [ping|port|full]"
        echo "Examples:"
        echo "  $0 192.168.1.0/24 ping    # Ping sweep only"
        echo "  $0 192.168.1.1 port       # Port scan single host"
        echo "  $0 192.168.1.0/24 full    # Full network scan"
        exit 1
        ;;
esac
```

Data Processing and Analysis

JSON Processor

```
#!/bin/bash
# json_processor.sh - JSON processing without external dependencies

# Simple JSON parser (basic implementation)
parse_json() {
    local json="$1"
    local key="$2"

    # Remove whitespace
    json=$(echo "$json" | tr -d ' \t\n\r')

    # Extract value for key
    local pattern="\"$key\":\"([^\"]*)\""
    if [[ $json =~ $pattern ]]; then
        echo "${BASH_REMATCH[1]}"
    else
        # Try numeric value
        pattern="\"$key\":([0-9]+)"
        if [[ $json =~ $pattern ]]; then
            echo "${BASH_REMATCH[1]}"
        fi
    fi
}

# Extract array elements
parse_json_array() {
    local json="$1"
    local array_key="$2"

    # Extract array content
    local pattern="\"$array_key\":\[([^\]]*)\]"
    if [[ $json =~ $pattern ]]; then
        local array_content="${BASH_REMATCH[1]}"

        # Split by comma and clean up
        IFS=',' read -ra elements <<< "$array_content"
        for element in "${elements[@]}"; do
            element=$(echo "$element" | sed 's/^[[:space:]]*"//;s/"[[:space:]]*$//')
            echo "$element"
        done
    fi
}
```

```
# Create JSON object
create_json() {
    local -n data_ref=$1
    local json="{"
    local first=true

    for key in "${!data_ref[@]}"; do
        if [ "$first" = true ]; then
            first=false
        else
            json+=","
        fi

        json+="\"$key\":\"${data_ref[$key]}\""
    done

    json+="}"
    echo "$json"
}

# Process JSON file
process_json_file() {
    local file="$1"
    local operation="$2"
    local key="$3"

    if [ ! -f "$file" ]; then
        echo "Error: File '$file' not found"
        return 1
    fi

    local json_content=$(cat "$file")

    case "$operation" in
        get)
            parse_json "$json_content" "$key"
            ;;
        array)
            parse_json_array "$json_content" "$key"
            ;;
        keys)
            # Extract all keys
            echo "$json_content" | grep -o '"[^"]*"[[:space:]]*:' | sed 's/"//g;s/[[:space:]
            ;;
        validate)
            # Basic JSON validation
```

```
            if echo "$json_content" | python3 -m json.tool >/dev/null 2>&1; then
                echo "Valid JSON"
            else
                echo "Invalid JSON"
            fi
            ;;
        pretty)
            # Pretty print JSON
            if command -v python3 >/dev/null 2>&1; then
                echo "$json_content" | python3 -m json.tool
            else
                echo "$json_content"
            fi
            ;;
        *)
            echo "Unknown operation: $operation"
            echo "Available operations: get, array, keys, validate, pretty"
            return 1
            ;;
    esac
}

# Example usage and testing
if [[ "${BASH_SOURCE[0]}" == "${0}" ]]; then
    # Create sample JSON file for testing
    cat > sample.json << 'EOF'
{
    "name": "John Doe",
    "age": 30,
    "city": "New York",
    "hobbies": ["reading", "swimming", "coding"],
    "active": true,
    "address": {
        "street": "123 Main St",
        "zip": "10001"
    }
}
EOF

    if [ $# -eq 0 ]; then
        echo "JSON Processor Demo"
        echo "==================="
        echo

        echo "Sample JSON content:"
        cat sample.json
        echo
```

```
        echo "Extracting 'name':"
        process_json_file sample.json get name
        echo

        echo "Extracting 'age':"
        process_json_file sample.json get age
        echo

        echo "Extracting 'hobbies' array:"
        process_json_file sample.json array hobbies
        echo

        echo "All keys:"
        process_json_file sample.json keys
        echo

        echo "Validating JSON:"
        process_json_file sample.json validate

    else
        process_json_file "$@"
    fi
fi
```

Data Aggregator

```
#!/bin/bash
# data_aggregator.sh - Aggregate and analyze data from multiple sources

declare -A metrics
declare -A counters
OUTPUT_FORMAT="${OUTPUT_FORMAT:-table}"

# Add metric
add_metric() {
    local key="$1"
    local value="$2"

    if [[ $value =~ ^[0-9]+(\.[0-9]+)?$ ]]; then
        metrics["$key"]="${metrics["$key"]:-0}"
        metrics["$key"]=$(echo "${metrics["$key"]} + $value" | bc -l)
        counters["$key"]=$((${counters["$key"]:-0} + 1))
    fi
}
```

```
# Process log file
process_log_file() {
    local file="$1"
    local format="$2"

    echo "Processing log file: $file (format: $format)"

    case "$format" in
        apache)
            # Apache access log format
            while IFS= read -r line; do
                # Extract response time if available
                if [[ $line =~ ([0-9]+)$ ]]; then
                    add_metric "response_time" "${BASH_REMATCH[1]}"
                fi

                # Extract status code
                if [[ $line =~ \"[^\"]*\"[[:space:]]+([0-9]{3}) ]]; then
                    add_metric "status_${BASH_REMATCH[1]}" 1
                fi

                # Extract bytes sent
                if [[ $line =~ [[:space:]]([0-9]+)[[:space:]]+\"[^\"]*\"[[:space:]]*$ ]]; th
                    add_metric "bytes_sent" "${BASH_REMATCH[1]}"
                fi
            done < "$file"
            ;;
        nginx)
            # Nginx access log format
            while IFS= read -r line; do
                # Similar processing for nginx logs
                if [[ $line =~ [[:space:]]([0-9]{3})[[:space:]] ]]; then
                    add_metric "status_${BASH_REMATCH[1]}" 1
                fi
            done < "$file"
            ;;
        csv)
            # CSV format - assume first line is header
            local headers=()
            local line_num=0

            while IFS=',' read -ra fields; do
                ((line_num++))

                if [ $line_num -eq 1 ]; then
                    headers=("${fields[@]}")
                else
```

```
                    for i in "${!fields[@]}"; do
                        local header="${headers[i]}"
                        local value="${fields[i]}"

                        if [[ $value =~ ^[0-9]+(\.[0-9]+)?$ ]]; then
                            add_metric "$header" "$value"
                        fi
                    done
                fi
            done < "$file"
            ;;
        *)
            echo "Unknown format: $format"
            return 1
            ;;
    esac
}

# Calculate statistics
calculate_stats() {
    declare -A averages
    declare -A totals

    for key in "${!metrics[@]}"; do
        totals["$key"]="${metrics["$key"]}"
        if [ "${counters["$key"]}" -gt 0 ]; then
            averages["$key"]=$(echo "scale=2; ${metrics["$key"]} / ${counters["$key"]}" | bc
        fi
    done

    case "$OUTPUT_FORMAT" in
        table)
            printf "%-20s %15s %15s %10s\n" "Metric" "Total" "Average" "Count"
            printf "%-20s %15s %15s %10s\n" "------" "-----" "-------" "-----"

            for key in $(printf '%s\n' "${!metrics[@]}" | sort); do
                printf "%-20s %15.2f %15.2f %10d\n" \
                    "$key" "${totals["$key"]}" "${averages["$key"]}" "${counters["$key"]}"
            done
            ;;
        json)
            echo "{"
            local first=true
            for key in "${!metrics[@]}"; do
                if [ "$first" = true ]; then
                    first=false
                else
```

```
                    echo -n ","
                fi
                printf '  "%s": {"total": %.2f, "average": %.2f, "count": %d}' \
                    "$key" "${totals["$key"]}" "${averages["$key"]}" "${counters["$key"]}"
            done
            echo
            echo "}"
            ;;
        csv)
            echo "metric,total,average,count"
            for key in "${!metrics[@]}"; do
                printf "%s,%.2f,%.2f,%d\n" \
                    "$key" "${totals["$key"]}" "${averages["$key"]}" "${counters["$key"]}"
            done
            ;;
    esac
}

# Generate report
generate_report() {
    local output_file="$1"

    {
        echo "Data Aggregation Report"
        echo "======================="
        echo "Generated: $(date)"
        echo "Files processed: ${#processed_files[@]}"
        echo

        calculate_stats

    } > "$output_file"

    echo "Report saved to: $output_file"
}

# Main processing function
main() {
    local files=("$@")
    local processed_files=()

    if [ ${#files[@]} -eq 0 ]; then
        echo "Usage: $0 <file1> [file2] ..."
        echo "Environment variables:"
        echo "  OUTPUT_FORMAT - Output format: table, json, csv (default: table)"
        exit 1
    fi
```

```
    for file in "${files[@]}"; do
        if [ ! -f "$file" ]; then
            echo "Warning: File '$file' not found, skipping"
            continue
        fi

        # Detect file format based on extension or content
        local format="apache"  # default
        case "$file" in
            *.csv) format="csv" ;;
            *nginx*) format="nginx" ;;
            *apache*) format="apache" ;;
        esac

        process_log_file "$file" "$format"
        processed_files+=("$file")
    done

    if [ ${#processed_files[@]} -eq 0 ]; then
        echo "No files were processed"
        exit 1
    fi

    echo
    echo "Aggregation Results:"
    echo "===================="
    calculate_stats

    # Generate detailed report
    local report_file="aggregation_report_$(date +%Y%m%d_%H%M%S).txt"
    generate_report "$report_file"
}

main "$@"
```

Security and Encryption

Password Manager

```
#!/bin/bash
# password_manager.sh - Simple encrypted password manager

VAULT_FILE="${VAULT_FILE:-$HOME/.password_vault.gpg}"
TEMP_FILE="/tmp/vault_temp.$$"
```

```
# Cleanup function
cleanup() {
    rm -f "$TEMP_FILE"
}
trap cleanup EXIT

# Check dependencies
check_dependencies() {
    if ! command -v gpg >/dev/null 2>&1; then
        echo "Error: GPG is required but not installed"
        exit 1
    fi
}

# Initialize vault
init_vault() {
    if [ -f "$VAULT_FILE" ]; then
        echo "Vault already exists at: $VAULT_FILE"
        return 1
    fi

    echo "Initializing new password vault..."
    echo "# Password Vault - Created $(date)" > "$TEMP_FILE"
    echo "# Format: service:username:password:notes" >> "$TEMP_FILE"

    encrypt_vault
    echo "Vault initialized at: $VAULT_FILE"
}

# Encrypt vault
encrypt_vault() {
    if ! gpg --symmetric --cipher-algo AES256 --output "$VAULT_FILE" "$TEMP_FILE"; then
        echo "Error: Failed to encrypt vault"
        exit 1
    fi
}

# Decrypt vault
decrypt_vault() {
    if [ ! -f "$VAULT_FILE" ]; then
        echo "Error: Vault file not found. Run 'init' first."
        exit 1
    fi

    if ! gpg --decrypt --output "$TEMP_FILE" "$VAULT_FILE" 2>/dev/null; then
        echo "Error: Failed to decrypt vault (wrong password?)"
```

```
        exit 1
    fi
}

# Add password entry
add_entry() {
    local service="$1"
    local username="$2"
    local password="$3"
    local notes="$4"

    if [ -z "$service" ] || [ -z "$username" ]; then
        echo "Error: Service and username are required"
        return 1
    fi

    decrypt_vault

    # Check if entry already exists
    if grep -q "^$service:$username:" "$TEMP_FILE"; then
        echo "Entry already exists. Use 'update' to modify."
        return 1
    fi

    # Generate password if not provided
    if [ -z "$password" ]; then
        password=$(generate_password)
        echo "Generated password: $password"
    fi

    echo "$service:$username:$password:$notes" >> "$TEMP_FILE"
    encrypt_vault

    echo "Entry added for $service ($username)"
}

# Generate secure password
generate_password() {
    local length="${1:-16}"
    tr -dc 'A-Za-z0-9!@#$%^&*' < /dev/urandom | head -c "$length"
}

# Search entries
search_entries() {
    local query="$1"

    decrypt_vault
```

```
    echo "Search results for: $query"
    echo "=========================="

    grep -i "$query" "$TEMP_FILE" | grep -v '^#' | while IFS=':' read -r service username pa
        echo "Service: $service"
        echo "Username: $username"
        echo "Password: [hidden - use 'show' to reveal]"
        echo "Notes: $notes"
        echo "---"
    done
}

# Show specific entry
show_entry() {
    local service="$1"
    local username="$2"

    decrypt_vault

    local entry=$(grep "^$service:$username:" "$TEMP_FILE")
    if [ -n "$entry" ]; then
        IFS=':' read -r service username password notes <<< "$entry"
        echo "Service: $service"
        echo "Username: $username"
        echo "Password: $password"
        echo "Notes: $notes"
    else
        echo "Entry not found: $service ($username)"
    fi
}

# List all entries
list_entries() {
    decrypt_vault

    echo "Password Vault Entries"
    echo "======================"

    grep -v '^#' "$TEMP_FILE" | while IFS=':' read -r service username password notes; do
        echo "$service ($username)"
    done
}

# Update entry
update_entry() {
    local service="$1"
```

```
    local username="$2"
    local new_password="$3"
    local new_notes="$4"

    decrypt_vault

    if ! grep -q "^$service:$username:" "$TEMP_FILE"; then
        echo "Entry not found: $service ($username)"
        return 1
    fi

    # Create updated file
    local updated_file="/tmp/vault_updated.$$"

    while IFS=':' read -r s u p n; do
        if [ "$s" = "$service" ] && [ "$u" = "$username" ]; then
            echo "$s:$u:${new_password:-$p}:${new_notes:-$n}"
        else
            echo "$s:$u:$p:$n"
        fi
    done < "$TEMP_FILE" > "$updated_file"

    mv "$updated_file" "$TEMP_FILE"
    encrypt_vault

    echo "Entry updated: $service ($username)"
}

# Delete entry
delete_entry() {
    local service="$1"
    local username="$2"

    decrypt_vault

    if ! grep -q "^$service:$username:" "$TEMP_FILE"; then
        echo "Entry not found: $service ($username)"
        return 1
    fi

    # Create file without the entry
    grep -v "^$service:$username:" "$TEMP_FILE" > "/tmp/vault_filtered.$$"
    mv "/tmp/vault_filtered.$$" "$TEMP_FILE"
    encrypt_vault

    echo "Entry deleted: $service ($username)"
}
```

```
# Show usage
show_usage() {
    cat << EOF
Password Manager - Encrypted password storage

Usage: $0 <command> [arguments]

Commands:
  init                            Initialize new vault
  add <service> <username> [password] [notes]  Add new entry
  show <service> <username>       Show specific entry
  search <query>                  Search entries
  list                            List all entries
  update <service> <username> [password] [notes]  Update entry
  delete <service> <username>     Delete entry
  generate [length]               Generate secure password

Environment Variables:
  VAULT_FILE                      Path to vault file (default: ~/.password_vault.gpg)

Examples:
  $0 init
  $0 add gmail john.doe@gmail.com
  $0 show gmail john.doe@gmail.com
  $0 search gmail
  $0 generate 20
EOF
}

# Main execution
check_dependencies

case "${1:-help}" in
    init)
        init_vault
        ;;
    add)
        add_entry "$2" "$3" "$4" "$5"
        ;;
    show)
        show_entry "$2" "$3"
        ;;
    search)
        search_entries "$2"
        ;;
    list)
```

```
        list_entries
        ;;
    update)
        update_entry "$2" "$3" "$4" "$5"
        ;;
    delete)
        delete_entry "$2" "$3"
        ;;
    generate)
        generate_password "$2"
        ;;
    help|--help|-h)
        show_usage
        ;;
    *)
        echo "Unknown command: $1"
        show_usage
        exit 1
        ;;
esac
```

These advanced examples demonstrate sophisticated Bash scripting techniques including process management, network programming, data processing, and security implementations. Each example showcases real-world applications and advanced programming patterns that can be adapted for various use cases.

Automation Examples

This section contains practical automation scripts for common system administration and development tasks.

System Administration Automation

Automated Backup System

```
#!/bin/bash
# automated_backup.sh - Comprehensive backup automation system

# Configuration
BACKUP_CONFIG="${BACKUP_CONFIG:-/etc/backup.conf}"
BACKUP_ROOT="${BACKUP_ROOT:-/backup}"
LOG_FILE="${LOG_FILE:-/var/log/backup.log}"
RETENTION_DAYS="${RETENTION_DAYS:-30}"
COMPRESSION="${COMPRESSION:-gzip}"
ENCRYPTION="${ENCRYPTION:-false}"
GPG_RECIPIENT="${GPG_RECIPIENT:-}"

# Default backup sources
declare -a BACKUP_SOURCES=(
    "/home"
    "/etc"
    "/var/www"
    "/opt"
)

# Logging function
log() {
    local level="$1"
    shift
    local message="$*"
    echo "$(date '+%Y-%m-%d %H:%M:%S') [$level] $message" | tee -a "$LOG_FILE"
}

# Load configuration
load_config() {
```

```
    if [ -f "$BACKUP_CONFIG" ]; then
        source "$BACKUP_CONFIG"
        log INFO "Configuration loaded from $BACKUP_CONFIG"
    else
        log WARN "Configuration file not found, using defaults"
    fi
}

# Create backup directory structure
setup_backup_dirs() {
    local timestamp=$(date +%Y%m%d_%H%M%S)
    BACKUP_DIR="$BACKUP_ROOT/$timestamp"

    mkdir -p "$BACKUP_DIR"
    log INFO "Created backup directory: $BACKUP_DIR"
}

# Backup database
backup_database() {
    local db_type="$1"
    local db_name="$2"
    local db_user="$3"
    local db_host="${4:-localhost}"

    log INFO "Starting database backup: $db_type/$db_name"

    case "$db_type" in
        mysql)
            if command -v mysqldump >/dev/null 2>&1; then
                mysqldump -h "$db_host" -u "$db_user" -p"$DB_PASSWORD" "$db_name" > "$BACKUP
                log INFO "MySQL database backup completed: $db_name"
            else
                log ERROR "mysqldump not found"
                return 1
            fi
            ;;
        postgresql)
            if command -v pg_dump >/dev/null 2>&1; then
                PGPASSWORD="$DB_PASSWORD" pg_dump -h "$db_host" -U "$db_user" "$db_name" > "
                log INFO "PostgreSQL database backup completed: $db_name"
            else
                log ERROR "pg_dump not found"
                return 1
            fi
            ;;
        *)
```

```
            log ERROR "Unsupported database type: $db_type"
            return 1
            ;;
    esac
}

# Backup files and directories
backup_files() {
    local source="$1"
    local exclude_file="/tmp/backup_exclude.$$"

    # Create exclude file
    cat > "$exclude_file" << EOF
*.tmp
*.log
*.cache
*~
.DS_Store
Thumbs.db
EOF

    if [ -d "$source" ]; then
        local backup_name=$(basename "$source")
        local archive_file="$BACKUP_DIR/${backup_name}.tar"

        log INFO "Backing up directory: $source"

        if tar --exclude-from="$exclude_file" -cf "$archive_file" "$source"; then
            # Compress if requested
            if [ "$COMPRESSION" = "gzip" ]; then
                gzip "$archive_file"
                archive_file="${archive_file}.gz"
            elif [ "$COMPRESSION" = "bzip2" ]; then
                bzip2 "$archive_file"
                archive_file="${archive_file}.bz2"
            fi

            # Encrypt if requested
            if [ "$ENCRYPTION" = "true" ] && [ -n "$GPG_RECIPIENT" ]; then
                gpg --trust-model always --encrypt -r "$GPG_RECIPIENT" "$archive_file"
                rm "$archive_file"
                archive_file="${archive_file}.gpg"
            fi

            log INFO "Backup completed: $archive_file"
        else
```

```
            log ERROR "Backup failed for: $source"
        fi
    else
        log WARN "Source not found: $source"
    fi

    rm -f "$exclude_file"
}

# Verify backup integrity
verify_backup() {
    local backup_file="$1"

    log INFO "Verifying backup: $backup_file"

    case "$backup_file" in
        *.tar.gz)
            if gzip -t "$backup_file" && tar -tzf "$backup_file" >/dev/null; then
                log INFO "Backup verification successful: $backup_file"
                return 0
            fi
            ;;
        *.tar.bz2)
            if bzip2 -t "$backup_file" && tar -tjf "$backup_file" >/dev/null; then
                log INFO "Backup verification successful: $backup_file"
                return 0
            fi
            ;;
        *.tar)
            if tar -tf "$backup_file" >/dev/null; then
                log INFO "Backup verification successful: $backup_file"
                return 0
            fi
            ;;
        *.gpg)
            if gpg --list-packets "$backup_file" >/dev/null 2>&1; then
                log INFO "Encrypted backup verification successful: $backup_file"
                return 0
            fi
            ;;
    esac

    log ERROR "Backup verification failed: $backup_file"
    return 1
}

# Clean old backups
```

```
cleanup_old_backups() {
    log INFO "Cleaning up backups older than $RETENTION_DAYS days"

    local deleted_count=0
    find "$BACKUP_ROOT" -maxdepth 1 -type d -name "20*" -mtime +$RETENTION_DAYS | while read
        log INFO "Removing old backup: $old_backup"
        rm -rf "$old_backup"
        ((deleted_count++))
    done

    log INFO "Cleanup completed, removed $deleted_count old backups"
}

# Send notification
send_notification() {
    local status="$1"
    local summary="$2"

    if [ -n "$NOTIFICATION_EMAIL" ] && command -v mail >/dev/null 2>&1; then
        {
            echo "Backup Status: $status"
            echo "Time: $(date)"
            echo "Summary: $summary"
            echo
            echo "Recent log entries:"
            tail -20 "$LOG_FILE"
        } | mail -s "Backup Report - $status" "$NOTIFICATION_EMAIL"

        log INFO "Notification sent to $NOTIFICATION_EMAIL"
    fi

    if [ -n "$SLACK_WEBHOOK" ] && command -v curl >/dev/null 2>&1; then
        local color="good"
        [ "$status" != "SUCCESS" ] && color="danger"

        curl -X POST -H 'Content-type: application/json' \
            --data "{\"attachments\":[{\"color\":\"$color\",\"title\":\"Backup Report\",\"te
            "$SLACK_WEBHOOK" >/dev/null 2>&1

        log INFO "Slack notification sent"
    fi
}

# Generate backup report
generate_report() {
    local report_file="$BACKUP_DIR/backup_report.txt"
    local total_size=0
```

```
    local file_count=0

    {
        echo "Backup Report"
        echo "============="
        echo "Date: $(date)"
        echo "Backup Directory: $BACKUP_DIR"
        echo

        echo "Backed up files:"
        echo "================"

        for file in "$BACKUP_DIR"/*; do
            if [ -f "$file" ] && [[ "$(basename "$file")" != "backup_report.txt" ]]; then
                local size=$(stat -c%s "$file" 2>/dev/null || stat -f%z "$file" 2>/dev/null)
                local human_size=$(numfmt --to=iec-i --suffix=B "$size")

                echo "$(basename "$file"): $human_size"
                total_size=$((total_size + size))
                ((file_count++))
            fi
        done

        echo
        echo "Summary:"
        echo "========"
        echo "Total files: $file_count"
        echo "Total size: $(numfmt --to=iec-i --suffix=B $total_size)"
        echo "Compression: $COMPRESSION"
        echo "Encryption: $ENCRYPTION"

    } > "$report_file"

    log INFO "Backup report generated: $report_file"
}

# Main backup process
main() {
    log INFO "Starting automated backup process"

    load_config
    setup_backup_dirs

    local backup_success=true
    local backed_up_items=0

    # Backup files and directories
```

```
for source in "${BACKUP_SOURCES[@]}"; do
    if backup_files "$source"; then
        ((backed_up_items++))
    else
        backup_success=false
    fi
done

# Backup databases if configured
if [ -n "$DATABASES" ]; then
    IFS=',' read -ra DB_LIST <<< "$DATABASES"
    for db_config in "${DB_LIST[@]}"; do
        IFS=':' read -ra DB_PARTS <<< "$db_config"
        if [ ${#DB_PARTS[@]} -ge 3 ]; then
            if backup_database "${DB_PARTS[0]}" "${DB_PARTS[1]}" "${DB_PARTS[2]}" "${DB_
                ((backed_up_items++))
            else
                backup_success=false
            fi
        fi
    done
fi

# Verify backups
for backup_file in "$BACKUP_DIR"/*; do
    if [ -f "$backup_file" ] && [[ "$(basename "$backup_file")" != "backup_report.txt" ]
        if ! verify_backup "$backup_file"; then
            backup_success=false
        fi
    fi
done

# Generate report
generate_report

# Cleanup old backups
cleanup_old_backups

# Send notification
if [ "$backup_success" = true ]; then
    send_notification "SUCCESS" "Backup completed successfully ($backed_up_items items)"
    log INFO "Backup process completed successfully"
else
    send_notification "FAILED" "Backup completed with errors"
    log ERROR "Backup process completed with errors"
    exit 1
fi
```

```
}

# Handle command line arguments
case "${1:-backup}" in
    backup)
        main
        ;;
    config)
        cat > "$BACKUP_CONFIG" << EOF
# Backup Configuration File
# Generated on $(date)

# Backup sources (space-separated)
BACKUP_SOURCES=(
    "/home"
    "/etc"
    "/var/www"
    "/opt"
)

# Database backups (format: type:name:user:host)
# DATABASES="mysql:myapp:dbuser:localhost,postgresql:webapp:pguser:localhost"

# Backup settings
RETENTION_DAYS=30
COMPRESSION="gzip"  # gzip, bzip2, or none
ENCRYPTION="false"
GPG_RECIPIENT=""

# Notifications
NOTIFICATION_EMAIL=""
SLACK_WEBHOOK=""

# Database credentials
DB_PASSWORD=""
EOF
        echo "Configuration template created: $BACKUP_CONFIG"
        ;;
    restore)
        echo "Restore functionality not implemented yet"
        ;;
    *)
        echo "Usage: $0 {backup|config|restore}"
        echo "Environment variables:"
        echo "  BACKUP_CONFIG    - Configuration file path"
        echo "  BACKUP_ROOT      - Root backup directory"
```

```
        echo "  LOG_FILE         - Log file path"
        exit 1
        ;;
esac
```

System Health Monitor

```
#!/bin/bash
# health_monitor.sh - Comprehensive system health monitoring

MONITOR_CONFIG="${MONITOR_CONFIG:-/etc/health_monitor.conf}"
LOG_FILE="${LOG_FILE:-/var/log/health_monitor.log}"
ALERT_LOG="${ALERT_LOG:-/var/log/health_alerts.log}"
CHECK_INTERVAL="${CHECK_INTERVAL:-300}"  # 5 minutes

# Thresholds
CPU_THRESHOLD="${CPU_THRESHOLD:-80}"
MEMORY_THRESHOLD="${MEMORY_THRESHOLD:-85}"
DISK_THRESHOLD="${DISK_THRESHOLD:-90}"
LOAD_THRESHOLD="${LOAD_THRESHOLD:-2.0}"
INODE_THRESHOLD="${INODE_THRESHOLD:-90}"

# Alert settings
ALERT_EMAIL="${ALERT_EMAIL:-}"
ALERT_WEBHOOK="${ALERT_WEBHOOK:-}"
ALERT_COOLDOWN="${ALERT_COOLDOWN:-3600}"  # 1 hour

# Services to monitor
CRITICAL_SERVICES=("ssh" "nginx" "apache2" "mysql" "postgresql")

# Logging function
log() {
    local level="$1"
    shift
    local message="$*"
    echo "$(date '+%Y-%m-%d %H:%M:%S') [$level] $message" | tee -a "$LOG_FILE"
}

# Alert function
alert() {
    local severity="$1"
    local message="$2"
    local alert_key="$3"

    # Check cooldown
```

```
    local last_alert_file="/tmp/alert_${alert_key}.last"
    if [ -f "$last_alert_file" ]; then
        local last_alert=$(cat "$last_alert_file")
        local current_time=$(date +%s)
        local time_diff=$((current_time - last_alert))

        if [ $time_diff -lt $ALERT_COOLDOWN ]; then
            log DEBUG "Alert suppressed due to cooldown: $alert_key"
            return
        fi
    fi

    # Log alert
    echo "$(date '+%Y-%m-%d %H:%M:%S') [$severity] $message" >> "$ALERT_LOG"
    log WARN "ALERT [$severity]: $message"

    # Send notifications
    send_alert_notification "$severity" "$message"

    # Update cooldown
    date +%s > "$last_alert_file"
}

# Send alert notifications
send_alert_notification() {
    local severity="$1"
    local message="$2"

    # Email notification
    if [ -n "$ALERT_EMAIL" ] && command -v mail >/dev/null 2>&1; then
        {
            echo "System Health Alert"
            echo "==================="
            echo "Severity: $severity"
            echo "Time: $(date)"
            echo "Host: $(hostname)"
            echo
            echo "Message: $message"
            echo
            echo "System Information:"
            echo "==================="
            uname -a
            echo
            echo "Load Average:"
            uptime
            echo
```

```
            echo "Memory Usage:"
            free -h
            echo
            echo "Disk Usage:"
            df -h
        } | mail -s "[$severity] System Health Alert - $(hostname)" "$ALERT_EMAIL"
    fi

    # Webhook notification
    if [ -n "$ALERT_WEBHOOK" ] && command -v curl >/dev/null 2>&1; then
        local color="warning"
        [ "$severity" = "CRITICAL" ] && color="danger"

        curl -X POST -H 'Content-type: application/json' \
            --data "{\"attachments\":[{\"color\":\"$color\",\"title\":\"System Health Alert\
            "$ALERT_WEBHOOK" >/dev/null 2>&1
    fi
}

# Check CPU usage
check_cpu() {
    local cpu_usage=$(top -bn1 | grep "Cpu(s)" | awk '{print $2}' | cut -d'%' -f1)
    cpu_usage=${cpu_usage%.*}

    log INFO "CPU usage: ${cpu_usage}%"

    if [ "$cpu_usage" -gt "$CPU_THRESHOLD" ]; then
        alert "WARNING" "High CPU usage: ${cpu_usage}% (threshold: ${CPU_THRESHOLD}%)" "cpu_

        # Get top CPU processes
        local top_processes=$(ps aux --sort=-%cpu | head -6 | tail -5 | awk '{print $11 " ("
        log INFO "Top CPU processes: $top_processes"
    fi

    echo "cpu_usage:$cpu_usage"
}

# Check memory usage
check_memory() {
    local memory_info=$(free | grep Mem)
    local total=$(echo $memory_info | awk '{print $2}')
    local used=$(echo $memory_info | awk '{print $3}')
    local available=$(echo $memory_info | awk '{print $7}')
    local memory_usage=$((used * 100 / total))

    log INFO "Memory usage: ${memory_usage}% (${used}/${total})"
```

```
    if [ "$memory_usage" -gt "$MEMORY_THRESHOLD" ]; then
        alert "WARNING" "High memory usage: ${memory_usage}% (threshold: ${MEMORY_THRESHOLD}

        # Get top memory processes
        local top_processes=$(ps aux --sort=-%mem | head -6 | tail -5 | awk '{print $11 " ("
        log INFO "Top memory processes: $top_processes"
    fi

    echo "memory_usage:$memory_usage"
    echo "memory_available:$available"
}

# Check disk usage
check_disk() {
    local alerts_sent=false

    df -h | grep -vE '^Filesystem|tmpfs|cdrom' | while read output; do
        local usage=$(echo $output | awk '{print $5}' | cut -d'%' -f1)
        local partition=$(echo $output | awk '{print $1}')
        local mount_point=$(echo $output | awk '{print $6}')
        local size=$(echo $output | awk '{print $2}')
        local available=$(echo $output | awk '{print $4}')

        log INFO "Disk usage $mount_point: ${usage}% ($available available)"

        if [ "$usage" -gt "$DISK_THRESHOLD" ]; then
            alert "WARNING" "High disk usage on $mount_point: ${usage}% (threshold: ${DISK_T
            alerts_sent=true
        fi

        echo "disk_usage_${mount_point//\//_}:$usage"
    done

    # Check inode usage
    df -i | grep -vE '^Filesystem|tmpfs|cdrom' | while read output; do
        local inode_usage=$(echo $output | awk '{print $5}' | cut -d'%' -f1)
        local mount_point=$(echo $output | awk '{print $6}')

        if [ "$inode_usage" -gt "$INODE_THRESHOLD" ]; then
            alert "WARNING" "High inode usage on $mount_point: ${inode_usage}% (threshold: $
        fi

        echo "inode_usage_${mount_point//\//_}:$inode_usage"
    done
}
```

```
# Check system load
check_load() {
    local load_1min=$(uptime | awk -F'load average:' '{print $2}' | awk '{print $1}' | tr -d
    local cpu_cores=$(nproc)
    local load_per_core=$(echo "scale=2; $load_1min / $cpu_cores" | bc -l)

    log INFO "Load average: $load_1min (${load_per_core} per core)"

    if (( $(echo "$load_per_core > $LOAD_THRESHOLD" | bc -l) )); then
        alert "WARNING" "High system load: $load_1min (${load_per_core} per core, threshold:
    fi

    echo "load_1min:$load_1min"
    echo "load_per_core:$load_per_core"
}

# Check critical services
check_services() {
    local failed_services=()

    for service in "${CRITICAL_SERVICES[@]}"; do
        if systemctl is-active --quiet "$service" 2>/dev/null; then
            log INFO "Service $service: running"
            echo "service_${service}:1"
        elif systemctl list-unit-files | grep -q "^$service.service"; then
            log WARN "Service $service: stopped"
            failed_services+=("$service")
            echo "service_${service}:0"
        else
            log DEBUG "Service $service: not installed"
            echo "service_${service}:-1"
        fi
    done

    if [ ${#failed_services[@]} -gt 0 ]; then
        alert "CRITICAL" "Critical services not running: ${failed_services[*]}" "services_do
    fi
}

# Check network connectivity
check_network() {
    local test_hosts=("8.8.8.8" "1.1.1.1")
    local failed_hosts=()

    for host in "${test_hosts[@]}"; do
        if ping -c 1 -W 5 "$host" >/dev/null 2>&1; then
            log INFO "Network connectivity to $host: OK"
```

```
            echo "network_${host//./_}:1"
        else
            log WARN "Network connectivity to $host: FAILED"
            failed_hosts+=("$host")
            echo "network_${host//./_}:0"
        fi
    done

    if [ ${#failed_hosts[@]} -gt 0 ]; then
        alert "CRITICAL" "Network connectivity issues: ${failed_hosts[*]}" "network_down"
    fi
}

# Check SSL certificates
check_ssl_certificates() {
    local cert_paths=("/etc/ssl/certs" "/etc/letsencrypt/live")
    local expiring_certs=()

    for cert_path in "${cert_paths[@]}"; do
        if [ -d "$cert_path" ]; then
            find "$cert_path" -name "*.crt" -o -name "cert.pem" | while read -r cert_file; d
                if [ -f "$cert_file" ]; then
                    local expiry_date=$(openssl x509 -in "$cert_file" -noout -enddate 2>/dev
                    if [ -n "$expiry_date" ]; then
                        local expiry_timestamp=$(date -d "$expiry_date" +%s 2>/dev/null)
                        local current_timestamp=$(date +%s)
                        local days_until_expiry=$(( (expiry_timestamp - current_timestamp) /

                        log INFO "SSL certificate $cert_file expires in $days_until_expiry d

                        if [ "$days_until_expiry" -lt 30 ]; then
                            expiring_certs+=("$(basename "$cert_file"):$days_until_expiry")
                        fi

                        echo "ssl_cert_$(basename "$cert_file" .crt | tr '.' '_')_days:$days
                    fi
                fi
            done
        fi
    done

    if [ ${#expiring_certs[@]} -gt 0 ]; then
        alert "WARNING" "SSL certificates expiring soon: ${expiring_certs[*]}" "ssl_expiring
    fi
}
```

```
# Check log files for errors
check_log_errors() {
    local log_files=("/var/log/syslog" "/var/log/auth.log" "/var/log/kern.log")
    local error_threshold=50
    local time_window="1 hour ago"

    for log_file in "${log_files[@]}"; do
        if [ -f "$log_file" ]; then
            local error_count=$(find "$log_file" -newermt "$time_window" -exec grep -i "erro

            log INFO "Errors in $(basename "$log_file"): $error_count (last hour)"

            if [ "$error_count" -gt "$error_threshold" ]; then
                alert "WARNING" "High error count in $log_file: $error_count errors in last
            fi

            echo "log_errors_$(basename "$log_file"):$error_count"
        fi
    done
}

# Generate metrics for monitoring systems
generate_metrics() {
    local metrics_file="/tmp/health_metrics.$$"

    {
        check_cpu
        check_memory
        check_disk
        check_load
        check_services
        check_network
        check_ssl_certificates
        check_log_errors
    } > "$metrics_file"

    # Output in Prometheus format if requested
    if [ "$OUTPUT_FORMAT" = "prometheus" ]; then
        while IFS=':' read -r metric value; do
            echo "system_${metric} ${value}"
        done < "$metrics_file"
    else
        cat "$metrics_file"
    fi

    rm -f "$metrics_file"
}
```

```
# Main monitoring loop
monitor_loop() {
    log INFO "Starting health monitoring loop (interval: ${CHECK_INTERVAL}s)"

    while true; do
        log INFO "Running health checks..."

        generate_metrics > "/tmp/current_metrics.txt"

        log INFO "Health check completed"

        sleep "$CHECK_INTERVAL"
    done
}

# One-time health check
health_check() {
    log INFO "Running one-time health check"
    generate_metrics
}

# Show system status dashboard
show_dashboard() {
    clear
    echo "System Health Dashboard - $(hostname)"
    echo "======================================"
    echo "Last updated: $(date)"
    echo

    # System info
    echo "System Information:"
    echo "=================="
    echo "Uptime: $(uptime -p)"
    echo "Kernel: $(uname -r)"
    echo "Architecture: $(uname -m)"
    echo

    # Current metrics
    echo "Current Metrics:"
    echo "==============="
    generate_metrics | while IFS=':' read -r metric value; do
        printf "%-30s: %s\n" "$metric" "$value"
    done

    echo
    echo "Recent Alerts:"
```

```
    echo "============="
    if [ -f "$ALERT_LOG" ]; then
        tail -5 "$ALERT_LOG"
    else
        echo "No recent alerts"
    fi
}

# Load configuration
load_config() {
    if [ -f "$MONITOR_CONFIG" ]; then
        source "$MONITOR_CONFIG"
        log INFO "Configuration loaded from $MONITOR_CONFIG"
    fi
}

# Main execution
load_config

case "${1:-check}" in
    check)
        health_check
        ;;
    monitor)
        monitor_loop
        ;;
    dashboard)
        show_dashboard
        ;;
    metrics)
        OUTPUT_FORMAT="prometheus" generate_metrics
        ;;
    config)
        cat > "$MONITOR_CONFIG" << EOF
# Health Monitor Configuration
# Generated on $(date)

# Check intervals (seconds)
CHECK_INTERVAL=300

# Thresholds
CPU_THRESHOLD=80
MEMORY_THRESHOLD=85
DISK_THRESHOLD=90
LOAD_THRESHOLD=2.0
INODE_THRESHOLD=90
```

```
# Alert settings
ALERT_EMAIL=""
ALERT_WEBHOOK=""
ALERT_COOLDOWN=3600

# Critical services to monitor
CRITICAL_SERVICES=("ssh" "nginx" "apache2" "mysql" "postgresql")

# Log files
LOG_FILE="/var/log/health_monitor.log"
ALERT_LOG="/var/log/health_alerts.log"
EOF
        echo "Configuration template created: $MONITOR_CONFIG"
        ;;
    *)
        echo "Usage: $0 {check|monitor|dashboard|metrics|config}"
        echo "Commands:"
        echo "  check      - Run one-time health check"
        echo "  monitor    - Start continuous monitoring"
        echo "  dashboard  - Show system status dashboard"
        echo "  metrics    - Output metrics in Prometheus format"
        echo "  config     - Generate configuration template"
        exit 1
        ;;
esac
```##
Development Automation

CI/CD Pipeline Script
```bash
#!/bin/bash
# cicd_pipeline.sh - Continuous Integration/Continuous Deployment pipeline

PROJECT_NAME="${PROJECT_NAME:-myapp}"
GIT_REPO="${GIT_REPO:-}"
BUILD_DIR="${BUILD_DIR:-./build}"
DEPLOY_DIR="${DEPLOY_DIR:-/var/www/$PROJECT_NAME}"
LOG_FILE="${LOG_FILE:-./pipeline.log}"

# Pipeline stages
STAGES=("checkout" "test" "build" "deploy" "notify")

# Configuration
SLACK_WEBHOOK="${SLACK_WEBHOOK:-}"
EMAIL_NOTIFICATIONS="${EMAIL_NOTIFICATIONS:-}"
DOCKER_REGISTRY="${DOCKER_REGISTRY:-}"
KUBERNETES_NAMESPACE="${KUBERNETES_NAMESPACE:-default}"
```

```
# Colors for output
RED='\033[0;31m'
GREEN='\033[0;32m'
YELLOW='\033[1;33m'
BLUE='\033[0;34m'
NC='\033[0m'

# Logging function
log() {
    local level="$1"
    shift
    local message="$*"
    local timestamp=$(date '+%Y-%m-%d %H:%M:%S')

    echo "[$timestamp] [$level] $message" | tee -a "$LOG_FILE"

    case "$level" in
        ERROR) echo -e "${RED}[$timestamp] ERROR: $message${NC}" ;;
        WARN)  echo -e "${YELLOW}[$timestamp] WARN:  $message${NC}" ;;
        INFO)  echo -e "${GREEN}[$timestamp] INFO:  $message${NC}" ;;
        DEBUG) echo -e "${BLUE}[$timestamp] DEBUG: $message${NC}" ;;
    esac
}

# Send notification
send_notification() {
    local status="$1"
    local stage="$2"
    local message="$3"

    # Slack notification
    if [ -n "$SLACK_WEBHOOK" ] && command -v curl >/dev/null 2>&1; then
        local color="good"
        [ "$status" != "SUCCESS" ] && color="danger"

        curl -X POST -H 'Content-type: application/json' \
            --data "{\"attachments\":[{\"color\":\"$color\",\"title\":\"Pipeline $status\",\
            "$SLACK_WEBHOOK" >/dev/null 2>&1
    fi

    # Email notification
    if [ -n "$EMAIL_NOTIFICATIONS" ] && command -v mail >/dev/null 2>&1; then
        {
            echo "Pipeline Status: $status"
            echo "Project: $PROJECT_NAME"
            echo "Stage: $stage"
```

```
            echo "Time: $(date)"
            echo
            echo "Message: $message"
            echo
            echo "Recent log entries:"
            tail -20 "$LOG_FILE"
        } | mail -s "Pipeline $status - $PROJECT_NAME" "$EMAIL_NOTIFICATIONS"
    fi
}

# Checkout stage
stage_checkout() {
    log INFO "Starting checkout stage"

    if [ -z "$GIT_REPO" ]; then
        log ERROR "GIT_REPO not specified"
        return 1
    fi

    # Clean previous build
    rm -rf "$BUILD_DIR"
    mkdir -p "$BUILD_DIR"

    # Clone repository
    if git clone "$GIT_REPO" "$BUILD_DIR"; then
        log INFO "Repository cloned successfully"

        # Get commit information
        cd "$BUILD_DIR"
        local commit_hash=$(git rev-parse HEAD)
        local commit_message=$(git log -1 --pretty=%B)
        local author=$(git log -1 --pretty=%an)

        log INFO "Commit: $commit_hash"
        log INFO "Author: $author"
        log INFO "Message: $commit_message"

        return 0
    else
        log ERROR "Failed to clone repository"
        return 1
    fi
}

# Test stage
stage_test() {
```

```
    log INFO "Starting test stage"

    cd "$BUILD_DIR"

    # Detect project type and run appropriate tests
    if [ -f "package.json" ]; then
        # Node.js project
        log INFO "Detected Node.js project"

        if command -v npm >/dev/null 2>&1; then
            log INFO "Installing dependencies..."
            if npm install; then
                log INFO "Dependencies installed"
            else
                log ERROR "Failed to install dependencies"
                return 1
            fi

            log INFO "Running tests..."
            if npm test; then
                log INFO "Tests passed"
            else
                log ERROR "Tests failed"
                return 1
            fi
        else
            log ERROR "npm not found"
            return 1
        fi

    elif [ -f "requirements.txt" ] || [ -f "setup.py" ]; then
        # Python project
        log INFO "Detected Python project"

        if command -v python3 >/dev/null 2>&1; then
            # Create virtual environment
            python3 -m venv venv
            source venv/bin/activate

            # Install dependencies
            if [ -f "requirements.txt" ]; then
                log INFO "Installing Python dependencies..."
                if pip install -r requirements.txt; then
                    log INFO "Dependencies installed"
                else
                    log ERROR "Failed to install dependencies"
                    return 1
```

```
                fi
            fi

            # Run tests
            if [ -f "pytest.ini" ] || [ -d "tests" ]; then
                log INFO "Running Python tests..."
                if python -m pytest; then
                    log INFO "Tests passed"
                else
                    log ERROR "Tests failed"
                    return 1
                fi
            fi
        else
            log ERROR "python3 not found"
            return 1
        fi

    elif [ -f "Makefile" ]; then
        # Make-based project
        log INFO "Detected Make-based project"

        if make test; then
            log INFO "Tests passed"
        else
            log ERROR "Tests failed"
            return 1
        fi

    else
        log WARN "No recognized test framework found, skipping tests"
    fi

    return 0
}

# Build stage
stage_build() {
    log INFO "Starting build stage"

    cd "$BUILD_DIR"

    # Detect project type and build
    if [ -f "Dockerfile" ]; then
        # Docker build
        log INFO "Building Docker image"
```

```
    local image_tag="$PROJECT_NAME:$(git rev-parse --short HEAD)"

    if docker build -t "$image_tag" .; then
        log INFO "Docker image built: $image_tag"

        # Push to registry if configured
        if [ -n "$DOCKER_REGISTRY" ]; then
            local registry_image="$DOCKER_REGISTRY/$image_tag"

            if docker tag "$image_tag" "$registry_image"; then
                log INFO "Tagged image for registry: $registry_image"

                if docker push "$registry_image"; then
                    log INFO "Image pushed to registry"
                else
                    log ERROR "Failed to push image to registry"
                    return 1
                fi
            fi
        fi
    else
        log ERROR "Docker build failed"
        return 1
    fi

elif [ -f "package.json" ]; then
    # Node.js build
    log INFO "Building Node.js application"

    if npm run build 2>/dev/null || npm run compile 2>/dev/null; then
        log INFO "Build completed"
    else
        log WARN "No build script found or build failed"
    fi

elif [ -f "Makefile" ]; then
    # Make build
    log INFO "Building with Make"

    if make; then
        log INFO "Build completed"
    else
        log ERROR "Build failed"
        return 1
    fi

else
```

```
        log INFO "No build process detected, copying files"

        # Simple file copy
        mkdir -p dist
        cp -r . dist/
        log INFO "Files copied to dist/"
    fi

    return 0
}

# Deploy stage
stage_deploy() {
    log INFO "Starting deploy stage"

    cd "$BUILD_DIR"

    # Backup current deployment
    if [ -d "$DEPLOY_DIR" ]; then
        local backup_dir="${DEPLOY_DIR}.backup.$(date +%Y%m%d_%H%M%S)"
        log INFO "Creating backup: $backup_dir"
        cp -r "$DEPLOY_DIR" "$backup_dir"
    fi

    # Deploy based on project type
    if [ -f "Dockerfile" ] && command -v kubectl >/dev/null 2>&1; then
        # Kubernetes deployment
        log INFO "Deploying to Kubernetes"

        local image_tag="$PROJECT_NAME:$(git rev-parse --short HEAD)"

        # Update deployment
        if kubectl set image deployment/$PROJECT_NAME container=$DOCKER_REGISTRY/$image_tag
            log INFO "Kubernetes deployment updated"

            # Wait for rollout
            if kubectl rollout status deployment/$PROJECT_NAME -n $KUBERNETES_NAMESPACE; the
                log INFO "Deployment rollout completed"
            else
                log ERROR "Deployment rollout failed"
                return 1
            fi
        else
            log ERROR "Failed to update Kubernetes deployment"
            return 1
        fi
```

```
    else
        # File-based deployment
        log INFO "Deploying files to $DEPLOY_DIR"

        # Create deploy directory
        mkdir -p "$DEPLOY_DIR"

        # Copy files
        if [ -d "dist" ]; then
            cp -r dist/* "$DEPLOY_DIR/"
        else
            cp -r . "$DEPLOY_DIR/"
        fi

        # Set permissions
        chown -R www-data:www-data "$DEPLOY_DIR" 2>/dev/null || true
        chmod -R 755 "$DEPLOY_DIR"

        # Restart services if needed
        if systemctl is-active --quiet nginx; then
            systemctl reload nginx
            log INFO "Nginx reloaded"
        fi

        if systemctl is-active --quiet apache2; then
            systemctl reload apache2
            log INFO "Apache reloaded"
        fi

        log INFO "Deployment completed"
    fi

    return 0
}

# Notify stage
stage_notify() {
    log INFO "Starting notify stage"

    local commit_hash=$(cd "$BUILD_DIR" && git rev-parse --short HEAD)
    local commit_message=$(cd "$BUILD_DIR" && git log -1 --pretty=%B | head -1)

    send_notification "SUCCESS" "deploy" "Deployment completed successfully. Commit: $commit

    return 0
}
```

```
# Run pipeline
run_pipeline() {
    local start_time=$(date +%s)
    log INFO "Starting CI/CD pipeline for $PROJECT_NAME"

    for stage in "${STAGES[@]}"; do
        log INFO "Running stage: $stage"

        local stage_start=$(date +%s)

        if "stage_$stage"; then
            local stage_end=$(date +%s)
            local stage_duration=$((stage_end - stage_start))
            log INFO "Stage $stage completed in ${stage_duration}s"
        else
            local stage_end=$(date +%s)
            local stage_duration=$((stage_end - stage_start))
            log ERROR "Stage $stage failed after ${stage_duration}s"

            send_notification "FAILED" "$stage" "Pipeline failed at stage: $stage"
            exit 1
        fi
    done

    local end_time=$(date +%s)
    local total_duration=$((end_time - start_time))

    log INFO "Pipeline completed successfully in ${total_duration}s"
}

# Rollback function
rollback() {
    local backup_dir="$1"

    if [ -z "$backup_dir" ]; then
        # Find latest backup
        backup_dir=$(ls -td "${DEPLOY_DIR}.backup."* 2>/dev/null | head -1)
    fi

    if [ -d "$backup_dir" ]; then
        log INFO "Rolling back to: $backup_dir"

        rm -rf "$DEPLOY_DIR"
        cp -r "$backup_dir" "$DEPLOY_DIR"

        log INFO "Rollback completed"
    else
```

```
        log ERROR "Backup directory not found: $backup_dir"
        exit 1
    fi
}

# Show pipeline status
show_status() {
    echo "CI/CD Pipeline Status"
    echo "====================="
    echo "Project: $PROJECT_NAME"
    echo "Build Directory: $BUILD_DIR"
    echo "Deploy Directory: $DEPLOY_DIR"
    echo

    if [ -f "$LOG_FILE" ]; then
        echo "Recent Activity:"
        echo "================"
        tail -20 "$LOG_FILE"
    fi

    echo
    echo "Available Backups:"
    echo "=================="
    ls -la "${DEPLOY_DIR}.backup."* 2>/dev/null || echo "No backups found"
}

# Main execution
case "${1:-run}" in
    run)
        run_pipeline
        ;;
    checkout)
        stage_checkout
        ;;
    test)
        stage_test
        ;;
    build)
        stage_build
        ;;
    deploy)
        stage_deploy
        ;;
    rollback)
        rollback "$2"
        ;;
    status)
```

```
        show_status
        ;;
    *)
        echo "Usage: $0 {run|checkout|test|build|deploy|rollback|status}"
        echo "Environment variables:"
        echo "  PROJECT_NAME         - Project name"
        echo "  GIT_REPO             - Git repository URL"
        echo "  BUILD_DIR            - Build directory"
        echo "  DEPLOY_DIR           - Deployment directory"
        echo "  SLACK_WEBHOOK        - Slack webhook URL"
        echo "  EMAIL_NOTIFICATIONS - Email for notifications"
        echo "  DOCKER_REGISTRY      - Docker registry URL"
        echo "  KUBERNETES_NAMESPACE - Kubernetes namespace"
        exit 1
        ;;
esac
```

Environment Provisioning Script

```
#!/bin/bash
# provision_environment.sh - Automated environment provisioning

ENVIRONMENT="${1:-development}"
CONFIG_DIR="${CONFIG_DIR:-./config}"
ANSIBLE_PLAYBOOK="${ANSIBLE_PLAYBOOK:-}"
TERRAFORM_DIR="${TERRAFORM_DIR:-./terraform}"

# Environment configurations
declare -A ENV_CONFIGS=(
    ["development"]="dev"
    ["staging"]="stage"
    ["production"]="prod"
)

# Required tools
REQUIRED_TOOLS=("ansible" "terraform" "kubectl" "docker")

# Logging function
log() {
    echo "$(date '+%Y-%m-%d %H:%M:%S') - $*"
}

# Check prerequisites
check_prerequisites() {
    log "Checking prerequisites..."
```

```
    local missing_tools=()

    for tool in "${REQUIRED_TOOLS[@]}"; do
        if ! command -v "$tool" >/dev/null 2>&1; then
            missing_tools+=("$tool")
        fi
    done

    if [ ${#missing_tools[@]} -gt 0 ]; then
        log "ERROR: Missing required tools: ${missing_tools[*]}"
        log "Please install the missing tools and try again."
        exit 1
    fi

    log "All prerequisites satisfied"
}

# Load environment configuration
load_environment_config() {
    local env="$1"
    local config_file="$CONFIG_DIR/${env}.conf"

    if [ -f "$config_file" ]; then
        source "$config_file"
        log "Loaded configuration for environment: $env"
    else
        log "WARNING: Configuration file not found: $config_file"
        log "Using default configuration"
    fi
}

# Provision infrastructure with Terraform
provision_infrastructure() {
    local env="$1"

    if [ ! -d "$TERRAFORM_DIR" ]; then
        log "Terraform directory not found: $TERRAFORM_DIR"
        return 1
    fi

    log "Provisioning infrastructure for environment: $env"

    cd "$TERRAFORM_DIR"

    # Initialize Terraform
    if terraform init; then
```

```
        log "Terraform initialized"
    else
        log "ERROR: Terraform initialization failed"
        return 1
    fi

    # Select workspace
    if terraform workspace select "$env" 2>/dev/null || terraform workspace new "$env"; then
        log "Terraform workspace: $env"
    else
        log "ERROR: Failed to select/create workspace: $env"
        return 1
    fi

    # Plan infrastructure changes
    log "Planning infrastructure changes..."
    if terraform plan -var-file="$CONFIG_DIR/${env}.tfvars" -out="$env.tfplan"; then
        log "Terraform plan completed"
    else
        log "ERROR: Terraform planning failed"
        return 1
    fi

    # Apply infrastructure changes
    log "Applying infrastructure changes..."
    if terraform apply "$env.tfplan"; then
        log "Infrastructure provisioned successfully"

        # Save outputs
        terraform output -json > "$CONFIG_DIR/${env}_outputs.json"
        log "Terraform outputs saved"
    else
        log "ERROR: Terraform apply failed"
        return 1
    fi

    cd - >/dev/null
}

# Configure servers with Ansible
configure_servers() {
    local env="$1"
    local playbook="${ANSIBLE_PLAYBOOK:-site.yml}"

    if [ ! -f "$playbook" ]; then
        log "Ansible playbook not found: $playbook"
        return 1
```

```
    fi

    log "Configuring servers for environment: $env"

    # Run Ansible playbook
    if ansible-playbook -i "inventory/${env}" "$playbook" --extra-vars "environment=$env"; t
        log "Server configuration completed"
    else
        log "ERROR: Ansible playbook execution failed"
        return 1
    fi
}

# Deploy applications
deploy_applications() {
    local env="$1"

    log "Deploying applications for environment: $env"

    # Deploy to Kubernetes if configured
    if [ -n "$KUBERNETES_NAMESPACE" ] && command -v kubectl >/dev/null 2>&1; then
        log "Deploying to Kubernetes namespace: $KUBERNETES_NAMESPACE"

        # Apply Kubernetes manifests
        if [ -d "k8s/$env" ]; then
            kubectl apply -f "k8s/$env/" -n "$KUBERNETES_NAMESPACE"
            log "Kubernetes manifests applied"
        fi

        # Wait for deployments to be ready
        kubectl wait --for=condition=available --timeout=300s deployment --all -n "$KUBERNET
        log "All deployments are ready"
    fi

    # Deploy with Docker Compose if configured
    if [ -f "docker-compose.${env}.yml" ]; then
        log "Deploying with Docker Compose"

        docker-compose -f "docker-compose.${env}.yml" up -d
        log "Docker Compose deployment completed"
    fi
}

# Setup monitoring and logging
setup_monitoring() {
    local env="$1"
```

```
    log "Setting up monitoring for environment: $env"

    # Deploy monitoring stack
    if [ -d "monitoring/$env" ]; then
        kubectl apply -f "monitoring/$env/" -n "monitoring-$env" 2>/dev/null || true
        log "Monitoring stack deployed"
    fi

    # Configure log aggregation
    if [ -f "logging/fluentd-${env}.yml" ]; then
        kubectl apply -f "logging/fluentd-${env}.yml" -n "logging-$env" 2>/dev/null || true
        log "Log aggregation configured"
    fi
}

# Run health checks
run_health_checks() {
    local env="$1"

    log "Running health checks for environment: $env"

    # Load health check endpoints from configuration
    local health_endpoints=()
    if [ -f "$CONFIG_DIR/${env}_health_checks.txt" ]; then
        while IFS= read -r endpoint; do
            health_endpoints+=("$endpoint")
        done < "$CONFIG_DIR/${env}_health_checks.txt"
    fi

    # Default health checks
    if [ ${#health_endpoints[@]} -eq 0 ]; then
        health_endpoints=("http://localhost:8080/health" "http://localhost:3000/health")
    fi

    local failed_checks=0

    for endpoint in "${health_endpoints[@]}"; do
        log "Checking health endpoint: $endpoint"

        if curl -f -s "$endpoint" >/dev/null; then
            log "  Health check passed: $endpoint"
        else
            log "  Health check failed: $endpoint"
            ((failed_checks++))
        fi
    done
```

```
    if [ $failed_checks -eq 0 ]; then
        log "All health checks passed"
        return 0
    else
        log "ERROR: $failed_checks health checks failed"
        return 1
    fi
}

# Generate environment documentation
generate_documentation() {
    local env="$1"
    local doc_file="docs/${env}_environment.md"

    log "Generating documentation for environment: $env"

    mkdir -p docs

    {
        echo "# $env Environment Documentation"
        echo
        echo "Generated on: $(date)"
        echo

        echo "## Infrastructure"
        if [ -f "$CONFIG_DIR/${env}_outputs.json" ]; then
            echo "### Terraform Outputs"
            echo '```json'
            cat "$CONFIG_DIR/${env}_outputs.json"
            echo '````'
        fi

        echo
        echo "## Services"
        if command -v kubectl >/dev/null 2>&1; then
            echo "### Kubernetes Services"
            echo '````'
            kubectl get services -n "$KUBERNETES_NAMESPACE" 2>/dev/null || echo "No Kubernet
            echo '````'
        fi

        echo
        echo "## Configuration"
        if [ -f "$CONFIG_DIR/${env}.conf" ]; then
            echo "### Environment Configuration"
            echo '```bash'
            cat "$CONFIG_DIR/${env}.conf"
```

```
            echo '```'
        fi

    } > "$doc_file"

    log "Documentation generated: $doc_file"
}

# Cleanup environment
cleanup_environment() {
    local env="$1"

    log "Cleaning up environment: $env"

    # Confirm cleanup
    read -p "Are you sure you want to destroy the $env environment? (yes/no): " confirm
    if [ "$confirm" != "yes" ]; then
        log "Cleanup cancelled"
        return 0
    fi

    # Destroy Kubernetes resources
    if [ -d "k8s/$env" ] && command -v kubectl >/dev/null 2>&1; then
        kubectl delete -f "k8s/$env/" -n "$KUBERNETES_NAMESPACE" 2>/dev/null || true
        log "Kubernetes resources deleted"
    fi

    # Destroy Docker Compose services
    if [ -f "docker-compose.${env}.yml" ]; then
        docker-compose -f "docker-compose.${env}.yml" down -v
        log "Docker Compose services stopped"
    fi

    # Destroy Terraform infrastructure
    if [ -d "$TERRAFORM_DIR" ]; then
        cd "$TERRAFORM_DIR"
        terraform workspace select "$env"
        terraform destroy -var-file="$CONFIG_DIR/${env}.tfvars" -auto-approve
        terraform workspace select default
        terraform workspace delete "$env"
        cd - >/dev/null
        log "Infrastructure destroyed"
    fi

    log "Environment cleanup completed"
}
```

```
# Show environment status
show_status() {
    local env="$1"

    echo "Environment Status: $env"
    echo "======================="
    echo

    # Terraform status
    if [ -d "$TERRAFORM_DIR" ]; then
        echo "Infrastructure Status:"
        cd "$TERRAFORM_DIR"
        terraform workspace select "$env" 2>/dev/null && terraform show -json | jq -r '.valu
        cd - >/dev/null
        echo
    fi

    # Kubernetes status
    if command -v kubectl >/dev/null 2>&1; then
        echo "Kubernetes Status:"
        kubectl get all -n "$KUBERNETES_NAMESPACE" 2>/dev/null || echo "No Kubernetes resour
        echo
    fi

    # Docker status
    if [ -f "docker-compose.${env}.yml" ]; then
        echo "Docker Compose Status:"
        docker-compose -f "docker-compose.${env}.yml" ps
    fi
}

# Main provisioning process
provision_environment() {
    local env="$1"

    log "Starting environment provisioning: $env"

    check_prerequisites
    load_environment_config "$env"

    # Provision infrastructure
    if provision_infrastructure "$env"; then
        log "Infrastructure provisioning completed"
    else
        log "ERROR: Infrastructure provisioning failed"
        exit 1
    fi
```

```
    # Configure servers
    if configure_servers "$env"; then
        log "Server configuration completed"
    else
        log "ERROR: Server configuration failed"
        exit 1
    fi

    # Deploy applications
    if deploy_applications "$env"; then
        log "Application deployment completed"
    else
        log "ERROR: Application deployment failed"
        exit 1
    fi

    # Setup monitoring
    setup_monitoring "$env"

    # Run health checks
    if run_health_checks "$env"; then
        log "Health checks passed"
    else
        log "WARNING: Some health checks failed"
    fi

    # Generate documentation
    generate_documentation "$env"

    log "Environment provisioning completed successfully: $env"
}

# Main execution
case "${2:-provision}" in
    provision)
        provision_environment "$ENVIRONMENT"
        ;;
    infrastructure)
        check_prerequisites
        load_environment_config "$ENVIRONMENT"
        provision_infrastructure "$ENVIRONMENT"
        ;;
    configure)
        check_prerequisites
        load_environment_config "$ENVIRONMENT"
        configure_servers "$ENVIRONMENT"
```

```
        ;;
    deploy)
        check_prerequisites
        load_environment_config "$ENVIRONMENT"
        deploy_applications "$ENVIRONMENT"
        ;;
    health)
        load_environment_config "$ENVIRONMENT"
        run_health_checks "$ENVIRONMENT"
        ;;
    status)
        show_status "$ENVIRONMENT"
        ;;
    cleanup)
        cleanup_environment "$ENVIRONMENT"
        ;;
    docs)
        generate_documentation "$ENVIRONMENT"
        ;;
    *)
        echo "Usage: $0 <environment> {provision|infrastructure|configure|deploy|health|stat
        echo
        echo "Environments: ${!ENV_CONFIGS[*]}"
        echo
        echo "Commands:"
        echo "  provision      - Full environment provisioning"
        echo "  infrastructure - Provision infrastructure only"
        echo "  configure      - Configure servers only"
        echo "  deploy         - Deploy applications only"
        echo "  health         - Run health checks"
        echo "  status         - Show environment status"
        echo "  cleanup        - Destroy environment"
        echo "  docs           - Generate documentation"
        exit 1
        ;;
esac
```

Web Scraping and API Automation

Web Scraper

```
#!/bin/bash
# web_scraper.sh - Web scraping and data extraction tool

URL="$1"
```

```
OUTPUT_DIR="${OUTPUT_DIR:-./scraped_data}"
USER_AGENT="${USER_AGENT:-Mozilla/5.0 (Linux; Bash Web Scraper)}"
DELAY="${DELAY:-1}"
MAX_PAGES="${MAX_PAGES:-10}"

# Create output directory
mkdir -p "$OUTPUT_DIR"

# Logging function
log() {
    echo "$(date '+%Y-%m-%d %H:%M:%S') - $*" | tee -a "$OUTPUT_DIR/scraper.log"
}

# Download page with retry logic
download_page() {
    local url="$1"
    local output_file="$2"
    local max_retries=3
    local retry_count=0

    while [ $retry_count -lt $max_retries ]; do
        if curl -s -L -A "$USER_AGENT" \
               -H "Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8"
               -H "Accept-Language: en-US,en;q=0.5" \
               -H "Accept-Encoding: gzip, deflate" \
               -H "Connection: keep-alive" \
               --compressed \
               "$url" > "$output_file"; then
            log "Downloaded: $url"
            return 0
        else
            ((retry_count++))
            log "Retry $retry_count/$max_retries for: $url"
            sleep $((retry_count * 2))
        fi
    done

    log "Failed to download after $max_retries retries: $url"
    return 1
}

# Extract links from HTML
extract_links() {
    local html_file="$1"
    local base_url="$2"

    # Extract all href attributes
```

```
    grep -oP 'href="\K[^"]*' "$html_file" | while read -r link; do
        # Convert relative URLs to absolute
        if [[ $link =~ ^https?:// ]]; then
            echo "$link"
        elif [[ $link =~ ^/ ]]; then
            echo "${base_url%/}$link"
        elif [[ $link =~ ^[^/] ]]; then
            echo "${base_url%/}/$link"
        fi
    done | sort -u
}

# Extract text content
extract_text() {
    local html_file="$1"
    local output_file="$2"

    # Remove HTML tags and extract text
    sed 's/<[^>]*>//g' "$html_file" | \
    sed 's/ / /g; s/&amp;/\&/g; s/&lt;/</g; s/&gt;/>/g; s/&quot;/"/g' | \
    sed '/^[[:space:]]*$/d' > "$output_file"

    log "Extracted text to: $output_file"
}

# Extract specific data using patterns
extract_data() {
    local html_file="$1"
    local pattern="$2"
    local output_file="$3"

    grep -oP "$pattern" "$html_file" > "$output_file"
    local count=$(wc -l < "$output_file")
    log "Extracted $count items matching pattern: $pattern"
}

# Extract emails
extract_emails() {
    local html_file="$1"
    local output_file="$2"

    grep -oE '[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}' "$html_file" | sort -u > "$out
    local count=$(wc -l < "$output_file")
    log "Extracted $count unique email addresses"
}

# Extract phone numbers
```

```
extract_phones() {
    local html_file="$1"
    local output_file="$2"

    # Various phone number patterns
    grep -oE '(\+?1[-.\s]?)?\(?[0-9]{3}\)?[-.\s]?[0-9]{3}[-.\s]?[0-9]{4}' "$html_file" | sor
    local count=$(wc -l < "$output_file")
    log "Extracted $count phone numbers"
}

# Extract images
extract_images() {
    local html_file="$1"
    local base_url="$2"
    local output_file="$3"

    grep -oP 'src="\K[^"]*\.(jpg|jpeg|png|gif|webp)' "$html_file" | while read -r img_src; d
        # Convert relative URLs to absolute
        if [[ $img_src =~ ^https?:// ]]; then
            echo "$img_src"
        elif [[ $img_src =~ ^/ ]]; then
            echo "${base_url%/}$img_src"
        elif [[ $img_src =~ ^[^/] ]]; then
            echo "${base_url%/}/$img_src"
        fi
    done | sort -u > "$output_file"

    local count=$(wc -l < "$output_file")
    log "Extracted $count image URLs"
}

# Download images
download_images() {
    local image_list="$1"
    local download_dir="$OUTPUT_DIR/images"

    mkdir -p "$download_dir"

    while IFS= read -r img_url; do
        local filename=$(basename "$img_url" | cut -d'?' -f1)
        local output_path="$download_dir/$filename"

        if [ ! -f "$output_path" ]; then
            if curl -s -L -A "$USER_AGENT" "$img_url" -o "$output_path"; then
                log "Downloaded image: $filename"
            else
```

```
                log "Failed to download image: $img_url"
            fi
            sleep "$DELAY"
        fi
    done < "$image_list"
}

# Generate sitemap
generate_sitemap() {
    local links_file="$1"
    local sitemap_file="$OUTPUT_DIR/sitemap.xml"

    {
        echo '<?xml version="1.0" encoding="UTF-8"?>'
        echo '<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">'

        while IFS= read -r url; do
            echo "  <url>"
            echo "    <loc>$url</loc>"
            echo "    <lastmod>$(date -I)</lastmod>"
            echo "  </url>"
        done < "$links_file"

        echo '</urlset>'
    } > "$sitemap_file"

    log "Generated sitemap: $sitemap_file"
}

# Analyze page structure
analyze_structure() {
    local html_file="$1"
    local analysis_file="$OUTPUT_DIR/structure_analysis.txt"

    {
        echo "Page Structure Analysis"
        echo "======================="
        echo "Generated: $(date)"
        echo

        echo "HTML Tags Count:"
        echo "================"
        grep -o '<[^>]*>' "$html_file" | sed 's/<\([^ >]*\).*/\1/' | sort | uniq -c | sort -

        echo
        echo "Meta Tags:"
        echo "=========="
```

```
        grep -i '<meta' "$html_file" || echo "No meta tags found"

        echo
        echo "Title:"
        echo "======"
        grep -oP '<title>\K[^<]*' "$html_file" || echo "No title found"

        echo
        echo "Headings:"
        echo "========="
        grep -oP '<h[1-6][^>]*>\K[^<]*' "$html_file" || echo "No headings found"

        echo
        echo "Forms:"
        echo "======"
        grep -c '<form' "$html_file" | xargs echo "Form count:"

        echo
        echo "Scripts:"
        echo "========"
        grep -c '<script' "$html_file" | xargs echo "Script count:"

        echo
        echo "Stylesheets:"
        echo "============"
        grep -c '<link.*stylesheet' "$html_file" | xargs echo "Stylesheet count:"

    } > "$analysis_file"

    log "Structure analysis saved: $analysis_file"
}

# Main scraping function
scrape_website() {
    local start_url="$1"
    local base_url=$(echo "$start_url" | sed 's|^\(https\?://[^/]*\).*|\1|')

    log "Starting web scraping: $start_url"
    log "Base URL: $base_url"

    local page_count=0
    local visited_file="$OUTPUT_DIR/visited_urls.txt"
    local queue_file="$OUTPUT_DIR/url_queue.txt"

    # Initialize queue
    echo "$start_url" > "$queue_file"
    touch "$visited_file"
```

```
while [ $page_count -lt $MAX_PAGES ] && [ -s "$queue_file" ]; do
    # Get next URL from queue
    local current_url=$(head -1 "$queue_file")
    sed -i '1d' "$queue_file"

    # Skip if already visited
    if grep -Fxq "$current_url" "$visited_file"; then
        continue
    fi

    # Mark as visited
    echo "$current_url" >> "$visited_file"

    ((page_count++))
    log "Processing page $page_count/$MAX_PAGES: $current_url"

    # Download page
    local html_file="$OUTPUT_DIR/page_${page_count}.html"
    if download_page "$current_url" "$html_file"; then
        # Extract data
        extract_text "$html_file" "$OUTPUT_DIR/page_${page_count}.txt"
        extract_emails "$html_file" "$OUTPUT_DIR/page_${page_count}_emails.txt"
        extract_phones "$html_file" "$OUTPUT_DIR/page_${page_count}_phones.txt"
        extract_images "$html_file" "$base_url" "$OUTPUT_DIR/page_${page_count}_images.t

        # Extract links for further crawling
        local links_file="$OUTPUT_DIR/page_${page_count}_links.txt"
        extract_links "$html_file" "$base_url" > "$links_file"

        # Add new links to queue (same domain only)
        while IFS= read -r link; do
            if [[ $link =~ ^$base_url ]] && ! grep -Fxq "$link" "$visited_file"; then
                echo "$link" >> "$queue_file"
            fi
        done < "$links_file"

        # Analyze page structure for first page
        if [ $page_count -eq 1 ]; then
            analyze_structure "$html_file"
        fi
    fi

    # Respect rate limiting
    sleep "$DELAY"
done
```

```
    log "Scraping completed. Processed $page_count pages."

    # Generate consolidated reports
    generate_reports
}

# Generate consolidated reports
generate_reports() {
    log "Generating consolidated reports..."

    # Consolidate all emails
    cat "$OUTPUT_DIR"/page_*_emails.txt 2>/dev/null | sort -u > "$OUTPUT_DIR/all_emails.txt"
    local email_count=$(wc -l < "$OUTPUT_DIR/all_emails.txt" 2>/dev/null || echo 0)
    log "Total unique emails found: $email_count"

    # Consolidate all phone numbers
    cat "$OUTPUT_DIR"/page_*_phones.txt 2>/dev/null | sort -u > "$OUTPUT_DIR/all_phones.txt"
    local phone_count=$(wc -l < "$OUTPUT_DIR/all_phones.txt" 2>/dev/null || echo 0)
    log "Total unique phone numbers found: $phone_count"

    # Consolidate all images
    cat "$OUTPUT_DIR"/page_*_images.txt 2>/dev/null | sort -u > "$OUTPUT_DIR/all_images.txt"
    local image_count=$(wc -l < "$OUTPUT_DIR/all_images.txt" 2>/dev/null || echo 0)
    log "Total unique images found: $image_count"

    # Consolidate all links
    cat "$OUTPUT_DIR"/page_*_links.txt 2>/dev/null | sort -u > "$OUTPUT_DIR/all_links.txt"
    local link_count=$(wc -l < "$OUTPUT_DIR/all_links.txt" 2>/dev/null || echo 0)
    log "Total unique links found: $link_count"

    # Generate sitemap
    if [ -f "$OUTPUT_DIR/all_links.txt" ]; then
        generate_sitemap "$OUTPUT_DIR/all_links.txt"
    fi

    # Generate summary report
    {
        echo "Web Scraping Summary Report"
        echo "==========================="
        echo "Generated: $(date)"
        echo "Target URL: $URL"
        echo "Pages processed: $(ls "$OUTPUT_DIR"/page_*.html 2>/dev/null | wc -l)"
        echo "Unique emails: $email_count"
        echo "Unique phones: $phone_count"
        echo "Unique images: $image_count"
        echo "Unique links: $link_count"
        echo
```

```
        echo "Output directory: $OUTPUT_DIR"
        echo
        echo "Files generated:"
        ls -la "$OUTPUT_DIR"
    } > "$OUTPUT_DIR/summary_report.txt"

    log "Summary report generated: $OUTPUT_DIR/summary_report.txt"
}

# Show usage
show_usage() {
    cat << EOF
Web Scraper - Extract data from websites

Usage: $0 <url> [options]

Options:
  -d, --delay SECONDS     Delay between requests (default: 1)
  -m, --max-pages NUM     Maximum pages to scrape (default: 10)
  -o, --output DIR        Output directory (default: ./scraped_data)
  -u, --user-agent STR    User agent string
  -i, --images            Download images
  -h, --help              Show this help

Environment Variables:
  OUTPUT_DIR              Output directory
  USER_AGENT              User agent string
  DELAY                   Delay between requests
  MAX_PAGES               Maximum pages to scrape

Examples:
  $0 https://example.com
  $0 https://example.com -d 2 -m 5
  $0 https://example.com --images
EOF
}

# Parse command line arguments
DOWNLOAD_IMAGES=false

while [[ $# -gt 0 ]]; do
    case $1 in
        -d|--delay)
            DELAY="$2"
            shift 2
            ;;
```

```
        -m|--max-pages)
            MAX_PAGES="$2"
            shift 2
            ;;
        -o|--output)
            OUTPUT_DIR="$2"
            shift 2
            ;;
        -u|--user-agent)
            USER_AGENT="$2"
            shift 2
            ;;
        -i|--images)
            DOWNLOAD_IMAGES=true
            shift
            ;;
        -h|--help)
            show_usage
            exit 0
            ;;
        -*)
            echo "Unknown option: $1"
            show_usage
            exit 1
            ;;
        *)
            if [ -z "$URL" ]; then
                URL="$1"
            fi
            shift
            ;;
    esac
done

# Main execution
if [ -z "$URL" ]; then
    echo "Error: URL is required"
    show_usage
    exit 1
fi

# Validate URL
if ! [[ $URL =~ ^https?:// ]]; then
    echo "Error: Invalid URL format. Must start with http:// or https://"
    exit 1
fi
```

```
# Check dependencies
if ! command -v curl >/dev/null 2>&1; then
    echo "Error: curl is required but not installed"
    exit 1
fi

# Start scraping
scrape_website "$URL"

# Download images if requested
if [ "$DOWNLOAD_IMAGES" = true ] && [ -f "$OUTPUT_DIR/all_images.txt" ]; then
    log "Downloading images..."
    download_images "$OUTPUT_DIR/all_images.txt"
fi

log "Web scraping completed successfully"
echo "Results saved in: $OUTPUT_DIR"
```

These automation examples demonstrate comprehensive solutions for common DevOps and system administration tasks. Each script includes error handling, logging, configuration management, and notification capabilities, making them suitable for production environments.

Utility Scripts

This section contains practical utility scripts for everyday tasks, system maintenance, and productivity enhancement.

File and Directory Utilities

Duplicate File Finder

```
#!/bin/bash
# find_duplicates.sh - Find and manage duplicate files

SEARCH_DIR="${1:-.}"
ACTION="${ACTION:-list}"
MIN_SIZE="${MIN_SIZE:-1}"
HASH_ALGORITHM="${HASH_ALGORITHM:-md5sum}"

# Temporary files
TEMP_DIR="/tmp/duplicate_finder.$$"
mkdir -p "$TEMP_DIR"

# Cleanup function
cleanup() {
    rm -rf "$TEMP_DIR"
}
trap cleanup EXIT

# Logging function
log() {
    echo "$(date '+%H:%M:%S') - $*"
}

# Calculate file hash
calculate_hash() {
    local file="$1"
    case "$HASH_ALGORITHM" in
        md5sum) md5sum "$file" | cut -d' ' -f1 ;;
        sha1sum) sha1sum "$file" | cut -d' ' -f1 ;;
        sha256sum) sha256sum "$file" | cut -d' ' -f1 ;;
        *) md5sum "$file" | cut -d' ' -f1 ;;
```

```
    esac
}

# Find files by size
find_by_size() {
    log "Finding files by size..."

    find "$SEARCH_DIR" -type f -size +${MIN_SIZE}c -exec stat -c "%s %n" {} \; | \
    sort -n > "$TEMP_DIR/files_by_size.txt"

    # Group files by size
    awk '{print $1}' "$TEMP_DIR/files_by_size.txt" | uniq -d > "$TEMP_DIR/duplicate_sizes.tx

    local duplicate_size_count=$(wc -l < "$TEMP_DIR/duplicate_sizes.txt")
    log "Found $duplicate_size_count different file sizes with potential duplicates"
}

# Find duplicates by hash
find_duplicates() {
    log "Calculating file hashes..."

    local processed=0
    local total=$(wc -l < "$TEMP_DIR/duplicate_sizes.txt")

    while IFS= read -r size; do
        ((processed++))
        echo -ne "\rProgress: $processed/$total"

        # Get all files of this size
        grep "^$size " "$TEMP_DIR/files_by_size.txt" | cut -d' ' -f2- | while IFS= read -r f
            if [ -f "$file" ]; then
                local hash=$(calculate_hash "$file")
                echo "$hash $size $file"
            fi
        done >> "$TEMP_DIR/files_with_hash.txt"
    done < "$TEMP_DIR/duplicate_sizes.txt"

    echo

    # Group by hash
    sort "$TEMP_DIR/files_with_hash.txt" > "$TEMP_DIR/sorted_hashes.txt"
    awk '{print $1}' "$TEMP_DIR/sorted_hashes.txt" | uniq -d > "$TEMP_DIR/duplicate_hashes.t

    local duplicate_hash_count=$(wc -l < "$TEMP_DIR/duplicate_hashes.txt")
    log "Found $duplicate_hash_count groups of duplicate files"
}
```

```
# List duplicate files
list_duplicates() {
    log "Listing duplicate files..."

    local group_num=0
    local total_duplicates=0
    local total_wasted_space=0

    while IFS= read -r hash; do
        ((group_num++))
        echo
        echo "Duplicate Group $group_num (Hash: $hash):"
        echo "========================================="

        local files=()
        local file_count=0
        local file_size=0

        grep "^$hash " "$TEMP_DIR/sorted_hashes.txt" | while IFS=' ' read -r h size file; do
            ((file_count++))
            file_size="$size"
            echo "$file_count. $file ($(numfmt --to=iec-i --suffix=B $size))"
            files+=("$file")
        done

        # Calculate wasted space (all but one file)
        local wasted=$((file_size * (file_count - 1)))
        total_wasted_space=$((total_wasted_space + wasted))
        total_duplicates=$((total_duplicates + file_count - 1))

        echo "Files in group: $file_count"
        echo "Wasted space: $(numfmt --to=iec-i --suffix=B $wasted)"

    done < "$TEMP_DIR/duplicate_hashes.txt"

    echo
    echo "Summary:"
    echo "========"
    echo "Duplicate groups: $group_num"
    echo "Total duplicate files: $total_duplicates"
    echo "Total wasted space: $(numfmt --to=iec-i --suffix=B $total_wasted_space)"
}

# Interactive duplicate removal
interactive_remove() {
    log "Starting interactive duplicate removal..."
```

```
    local group_num=0
    local removed_count=0
    local freed_space=0

    while IFS= read -r hash; do
        ((group_num++))
        echo
        echo "Duplicate Group $group_num (Hash: $hash):"
        echo "========================================"

        local files=()
        local file_count=0
        local file_size=0

        # Collect files in this group
        while IFS=' ' read -r h size file; do
            if [ "$h" = "$hash" ]; then
                ((file_count++))
                file_size="$size"
                files+=("$file")
                echo "$file_count. $file"
                echo "   Size: $(numfmt --to=iec-i --suffix=B $size)"
                echo "   Modified: $(stat -c %y "$file")"
            fi
        done < "$TEMP_DIR/sorted_hashes.txt"

        echo
        echo "Choose action:"
        echo "k) Keep all files"
        echo "1-$file_count) Keep only file number X (delete others)"
        echo "a) Auto-keep oldest file"
        echo "s) Skip this group"

        read -p "Your choice: " choice

        case "$choice" in
            k|K)
                echo "Keeping all files in this group"
                ;;
            [1-9]*)
                if [ "$choice" -ge 1 ] && [ "$choice" -le $file_count ]; then
                    local keep_index=$((choice - 1))
                    echo "Keeping: ${files[$keep_index]}"

                    for i in "${!files[@]}"; do
                        if [ $i -ne $keep_index ]; then
                            echo "Removing: ${files[$i]}"
```

```
                        if rm "${files[$i]}"; then
                            ((removed_count++))
                            freed_space=$((freed_space + file_size))
                        fi
                    fi
                done
            else
                echo "Invalid choice, skipping group"
            fi
            ;;
        a|A)
            # Keep oldest file
            local oldest_file=""
            local oldest_time=0

            for file in "${files[@]}"; do
                local mtime=$(stat -c %Y "$file")
                if [ $oldest_time -eq 0 ] || [ $mtime -lt $oldest_time ]; then
                    oldest_time=$mtime
                    oldest_file="$file"
                fi
            done

            echo "Auto-keeping oldest file: $oldest_file"

            for file in "${files[@]}"; do
                if [ "$file" != "$oldest_file" ]; then
                    echo "Removing: $file"
                    if rm "$file"; then
                        ((removed_count++))
                        freed_space=$((freed_space + file_size))
                    fi
                fi
            done
            ;;
        s|S)
            echo "Skipping this group"
            ;;
        *)
            echo "Invalid choice, skipping group"
            ;;
    esac

done < "$TEMP_DIR/duplicate_hashes.txt"

echo
```

```
    echo "Removal Summary:"
    echo "================"
    echo "Files removed: $removed_count"
    echo "Space freed: $(numfmt --to=iec-i --suffix=B $freed_space)"
}

# Auto-remove duplicates (keep oldest)
auto_remove() {
    log "Auto-removing duplicates (keeping oldest files)..."

    local removed_count=0
    local freed_space=0

    while IFS= read -r hash; do
        local files=()
        local file_size=0

        # Collect files in this group
        while IFS=' ' read -r h size file; do
            if [ "$h" = "$hash" ]; then
                files+=("$file")
                file_size="$size"
            fi
        done < "$TEMP_DIR/sorted_hashes.txt"

        # Find oldest file
        local oldest_file=""
        local oldest_time=0

        for file in "${files[@]}"; do
            local mtime=$(stat -c %Y "$file")
            if [ $oldest_time -eq 0 ] || [ $mtime -lt $oldest_time ]; then
                oldest_time=$mtime
                oldest_file="$file"
            fi
        done

        echo "Keeping oldest: $oldest_file"

        # Remove other files
        for file in "${files[@]}"; do
            if [ "$file" != "$oldest_file" ]; then
                echo "Removing: $file"
                if rm "$file"; then
                    ((removed_count++))
                    freed_space=$((freed_space + file_size))
                fi
```

```
            fi
        done

    done < "$TEMP_DIR/duplicate_hashes.txt"

    echo
    echo "Auto-removal Summary:"
    echo "====================="
    echo "Files removed: $removed_count"
    echo "Space freed: $(numfmt --to=iec-i --suffix=B $freed_space)"
}

# Generate report
generate_report() {
    local report_file="duplicate_report_$(date +%Y%m%d_%H%M%S).txt"

    {
        echo "Duplicate Files Report"
        echo "======================"
        echo "Generated: $(date)"
        echo "Search directory: $SEARCH_DIR"
        echo "Minimum file size: $MIN_SIZE bytes"
        echo "Hash algorithm: $HASH_ALGORITHM"
        echo

        list_duplicates

    } > "$report_file"

    log "Report saved: $report_file"
}

# Main execution
main() {
    if [ ! -d "$SEARCH_DIR" ]; then
        echo "Error: Directory '$SEARCH_DIR' does not exist"
        exit 1
    fi

    log "Starting duplicate file search in: $SEARCH_DIR"
    log "Minimum file size: $MIN_SIZE bytes"
    log "Hash algorithm: $HASH_ALGORITHM"

    find_by_size
    find_duplicates

    case "$ACTION" in
```

```
        list)
            list_duplicates
            ;;
        interactive)
            interactive_remove
            ;;
        auto)
            auto_remove
            ;;
        report)
            generate_report
            ;;
        *)
            echo "Unknown action: $ACTION"
            echo "Available actions: list, interactive, auto, report"
            exit 1
            ;;
    esac
}

# Show usage
show_usage() {
    cat << EOF
Duplicate File Finder - Find and manage duplicate files

Usage: $0 [directory] [options]

Options:
  -a, --action ACTION     Action to perform: list, interactive, auto, report
  -s, --min-size SIZE     Minimum file size in bytes (default: 1)
  -h, --hash ALGORITHM    Hash algorithm: md5sum, sha1sum, sha256sum
  --help                  Show this help

Environment Variables:
  ACTION                  Action to perform
  MIN_SIZE                Minimum file size
  HASH_ALGORITHM          Hash algorithm

Actions:
  list                    List duplicate files (default)
  interactive             Interactive duplicate removal
  auto                    Auto-remove duplicates (keep oldest)
  report                  Generate detailed report

Examples:
  $0 /home/user/Documents
  $0 /home/user/Pictures --action interactive
```

```
  ACTION=auto MIN_SIZE=1024 $0 /var/log
EOF
}

# Parse command line arguments
while [[ $# -gt 0 ]]; do
    case $1 in
        -a|--action)
            ACTION="$2"
            shift 2
            ;;
        -s|--min-size)
            MIN_SIZE="$2"
            shift 2
            ;;
        -h|--hash)
            HASH_ALGORITHM="$2"
            shift 2
            ;;
        --help)
            show_usage
            exit 0
            ;;
        -*)
            echo "Unknown option: $1"
            show_usage
            exit 1
            ;;
        *)
            if [ -z "$SEARCH_DIR" ] || [ "$SEARCH_DIR" = "." ]; then
                SEARCH_DIR="$1"
            fi
            shift
            ;;
    esac
done

main
```

Directory Synchronizer

```
#!/bin/bash
# sync_directories.sh - Synchronize directories with various options

SOURCE_DIR="$1"
```

```
TARGET_DIR="$2"
SYNC_MODE="${SYNC_MODE:-mirror}"
DRY_RUN="${DRY_RUN:-false}"
EXCLUDE_FILE="${EXCLUDE_FILE:-}"
LOG_FILE="${LOG_FILE:-sync_$(date +%Y%m%d_%H%M%S).log}"

# Sync statistics
declare -i files_copied=0
declare -i files_updated=0
declare -i files_deleted=0
declare -i dirs_created=0
declare -i bytes_transferred=0

# Logging function
log() {
    local level="$1"
    shift
    local message="$*"
    echo "$(date '+%Y-%m-%d %H:%M:%S') [$level] $message" | tee -a "$LOG_FILE"
}

# Check if file should be excluded
is_excluded() {
    local file="$1"

    if [ -n "$EXCLUDE_FILE" ] && [ -f "$EXCLUDE_FILE" ]; then
        while IFS= read -r pattern; do
            # Skip comments and empty lines
            [[ $pattern =~ ^[[:space:]]*# ]] && continue
            [[ -z $pattern ]] && continue

            if [[ $file =~ $pattern ]]; then
                return 0  # File is excluded
            fi
        done < "$EXCLUDE_FILE"
    fi

    return 1  # File is not excluded
}

# Copy file with verification
copy_file() {
    local src="$1"
    local dst="$2"
    local action="$3"  # "copy" or "update"

    if [ "$DRY_RUN" = "true" ]; then
```

```
        log INFO "DRY RUN: Would $action $src -> $dst"
        return 0
    fi

    # Create destination directory if needed
    local dst_dir=$(dirname "$dst")
    if [ ! -d "$dst_dir" ]; then
        if mkdir -p "$dst_dir"; then
            log INFO "Created directory: $dst_dir"
            ((dirs_created++))
        else
            log ERROR "Failed to create directory: $dst_dir"
            return 1
        fi
    fi

    # Copy file
    if cp -p "$src" "$dst"; then
        local file_size=$(stat -c%s "$src" 2>/dev/null || stat -f%z "$src" 2>/dev/null)
        bytes_transferred=$((bytes_transferred + file_size))

        if [ "$action" = "copy" ]; then
            log INFO "Copied: $src -> $dst ($(numfmt --to=iec-i --suffix=B $file_size))"
            ((files_copied++))
        else
            log INFO "Updated: $src -> $dst ($(numfmt --to=iec-i --suffix=B $file_size))"
            ((files_updated++))
        fi
        return 0
    else
        log ERROR "Failed to $action: $src -> $dst"
        return 1
    fi
}

# Delete file or directory
delete_item() {
    local item="$1"

    if [ "$DRY_RUN" = "true" ]; then
        log INFO "DRY RUN: Would delete $item"
        return 0
    fi

    if rm -rf "$item"; then
        log INFO "Deleted: $item"
        ((files_deleted++))
```

```
        return 0
    else
        log ERROR "Failed to delete: $item"
        return 1
    fi
}

# Compare files
files_different() {
    local src="$1"
    local dst="$2"

    # Check if destination exists
    [ ! -f "$dst" ] && return 0

    # Compare modification times
    local src_mtime=$(stat -c %Y "$src" 2>/dev/null || stat -f %m "$src" 2>/dev/null)
    local dst_mtime=$(stat -c %Y "$dst" 2>/dev/null || stat -f %m "$dst" 2>/dev/null)

    [ "$src_mtime" -gt "$dst_mtime" ] && return 0

    # Compare sizes
    local src_size=$(stat -c %s "$src" 2>/dev/null || stat -f %z "$src" 2>/dev/null)
    local dst_size=$(stat -c %s "$dst" 2>/dev/null || stat -f %z "$dst" 2>/dev/null)

    [ "$src_size" -ne "$dst_size" ] && return 0

    return 1  # Files are the same
}

# Mirror sync (make target identical to source)
sync_mirror() {
    log INFO "Starting mirror synchronization"
    log INFO "Source: $SOURCE_DIR"
    log INFO "Target: $TARGET_DIR"

    # First pass: copy/update files from source to target
    find "$SOURCE_DIR" -type f | while IFS= read -r src_file; do
        # Skip excluded files
        if is_excluded "$src_file"; then
            log DEBUG "Excluded: $src_file"
            continue
        fi

        # Calculate relative path
        local rel_path="${src_file#$SOURCE_DIR/}"
        local dst_file="$TARGET_DIR/$rel_path"
```

```
        if files_different "$src_file" "$dst_file"; then
            if [ -f "$dst_file" ]; then
                copy_file "$src_file" "$dst_file" "update"
            else
                copy_file "$src_file" "$dst_file" "copy"
            fi
        fi
    done

    # Second pass: remove files from target that don't exist in source
    if [ -d "$TARGET_DIR" ]; then
        find "$TARGET_DIR" -type f | while IFS= read -r dst_file; do
            local rel_path="${dst_file#$TARGET_DIR/}"
            local src_file="$SOURCE_DIR/$rel_path"

            if [ ! -f "$src_file" ]; then
                delete_item "$dst_file"
            fi
        done

        # Remove empty directories
        find "$TARGET_DIR" -type d -empty -delete 2>/dev/null || true
    fi
}

# One-way sync (only copy new/updated files)
sync_oneway() {
    log INFO "Starting one-way synchronization"
    log INFO "Source: $SOURCE_DIR"
    log INFO "Target: $TARGET_DIR"

    find "$SOURCE_DIR" -type f | while IFS= read -r src_file; do
        # Skip excluded files
        if is_excluded "$src_file"; then
            log DEBUG "Excluded: $src_file"
            continue
        fi

        # Calculate relative path
        local rel_path="${src_file#$SOURCE_DIR/}"
        local dst_file="$TARGET_DIR/$rel_path"

        if files_different "$src_file" "$dst_file"; then
            if [ -f "$dst_file" ]; then
                copy_file "$src_file" "$dst_file" "update"
            else
```

```
                copy_file "$src_file" "$dst_file" "copy"
            fi
        fi
    done
}

# Bidirectional sync (sync both ways based on modification time)
sync_bidirectional() {
    log INFO "Starting bidirectional synchronization"
    log INFO "Directory A: $SOURCE_DIR"
    log INFO "Directory B: $TARGET_DIR"

    # Create list of all files in both directories
    local all_files_file="/tmp/sync_all_files.$$"

    {
        find "$SOURCE_DIR" -type f | sed "s|^$SOURCE_DIR/||"
        find "$TARGET_DIR" -type f | sed "s|^$TARGET_DIR/||"
    } | sort -u > "$all_files_file"

    while IFS= read -r rel_path; do
        # Skip excluded files
        if is_excluded "$rel_path"; then
            log DEBUG "Excluded: $rel_path"
            continue
        fi

        local file_a="$SOURCE_DIR/$rel_path"
        local file_b="$TARGET_DIR/$rel_path"

        if [ -f "$file_a" ] && [ -f "$file_b" ]; then
            # Both files exist, sync newer to older
            local mtime_a=$(stat -c %Y "$file_a" 2>/dev/null || stat -f %m "$file_a" 2>/dev/
            local mtime_b=$(stat -c %Y "$file_b" 2>/dev/null || stat -f %m "$file_b" 2>/dev/

            if [ "$mtime_a" -gt "$mtime_b" ]; then
                copy_file "$file_a" "$file_b" "update"
            elif [ "$mtime_b" -gt "$mtime_a" ]; then
                copy_file "$file_b" "$file_a" "update"
            fi
        elif [ -f "$file_a" ]; then
            # File only exists in A, copy to B
            copy_file "$file_a" "$file_b" "copy"
        elif [ -f "$file_b" ]; then
            # File only exists in B, copy to A
            copy_file "$file_b" "$file_a" "copy"
        fi
```

```
    done < "$all_files_file"

    rm -f "$all_files_file"
}

# Generate sync report
generate_report() {
    local report_file="sync_report_$(date +%Y%m%d_%H%M%S).txt"

    {
        echo "Directory Synchronization Report"
        echo "================================"
        echo "Generated: $(date)"
        echo "Source: $SOURCE_DIR"
        echo "Target: $TARGET_DIR"
        echo "Sync Mode: $SYNC_MODE"
        echo "Dry Run: $DRY_RUN"
        echo

        echo "Statistics:"
        echo "==========="
        echo "Files copied: $files_copied"
        echo "Files updated: $files_updated"
        echo "Files deleted: $files_deleted"
        echo "Directories created: $dirs_created"
        echo "Bytes transferred: $(numfmt --to=iec-i --suffix=B $bytes_transferred)"
        echo

        echo "Log file: $LOG_FILE"

        if [ -f "$EXCLUDE_FILE" ]; then
            echo
            echo "Exclusion patterns:"
            echo "==================="
            cat "$EXCLUDE_FILE"
        fi

    } > "$report_file"

    log INFO "Report generated: $report_file"
}

# Validate directories
validate_directories() {
    if [ ! -d "$SOURCE_DIR" ]; then
        log ERROR "Source directory does not exist: $SOURCE_DIR"
        exit 1
```

```
    fi

    if [ "$SYNC_MODE" != "bidirectional" ] && [ ! -d "$TARGET_DIR" ]; then
        log INFO "Creating target directory: $TARGET_DIR"
        if ! mkdir -p "$TARGET_DIR"; then
            log ERROR "Failed to create target directory: $TARGET_DIR"
            exit 1
        fi
    fi

    # Check for circular sync (target inside source or vice versa)
    local source_real=$(realpath "$SOURCE_DIR")
    local target_real=$(realpath "$TARGET_DIR" 2>/dev/null || echo "$TARGET_DIR")

    if [[ "$target_real" == "$source_real"* ]] || [[ "$source_real" == "$target_real"* ]]; t
        log ERROR "Circular sync detected: target cannot be inside source or vice versa"
        exit 1
    fi
}

# Show progress
show_progress() {
    if [ "$DRY_RUN" != "true" ]; then
        while true; do
            echo -ne "\rFiles: $files_copied copied, $files_updated updated, $files_deleted
            sleep 1
        done &
        local progress_pid=$!

        # Kill progress display when script exits
        trap "kill $progress_pid 2>/dev/null" EXIT
    fi
}

# Main synchronization function
main() {
    log INFO "Starting directory synchronization"

    validate_directories

    if [ "$DRY_RUN" = "true" ]; then
        log INFO "DRY RUN MODE - No files will be modified"
    fi

    show_progress

    case "$SYNC_MODE" in
```

```
        mirror)
            sync_mirror
            ;;
        oneway)
            sync_oneway
            ;;
        bidirectional)
            sync_bidirectional
            ;;
        *)
            log ERROR "Unknown sync mode: $SYNC_MODE"
            exit 1
            ;;
    esac

    log INFO "Synchronization completed"

    echo
    echo "Synchronization Summary:"
    echo "========================"
    echo "Files copied: $files_copied"
    echo "Files updated: $files_updated"
    echo "Files deleted: $files_deleted"
    echo "Directories created: $dirs_created"
    echo "Bytes transferred: $(numfmt --to=iec-i --suffix=B $bytes_transferred)"

    generate_report
}

# Show usage
show_usage() {
    cat << EOF
Directory Synchronizer - Synchronize directories with various options

Usage: $0 <source_dir> <target_dir> [options]

Options:
  -m, --mode MODE         Sync mode: mirror, oneway, bidirectional
  -n, --dry-run           Show what would be done without making changes
  -e, --exclude FILE      File containing exclusion patterns
  -l, --log FILE          Log file path
  --help                  Show this help

Environment Variables:
  SYNC_MODE               Sync mode
  DRY_RUN                 Dry run mode (true/false)
  EXCLUDE_FILE            Exclusion patterns file
```

```
  LOG_FILE              Log file path

Sync Modes:
  mirror                Make target identical to source (default)
  oneway                Copy new/updated files from source to target
  bidirectional         Sync both ways based on modification time

Examples:
  $0 /home/user/docs /backup/docs
  $0 /home/user/docs /backup/docs --mode oneway --dry-run
  SYNC_MODE=bidirectional $0 /home/user/docs /shared/docs
EOF
}

# Parse command line arguments
while [[ $# -gt 0 ]]; do
    case $1 in
        -m|--mode)
            SYNC_MODE="$2"
            shift 2
            ;;
        -n|--dry-run)
            DRY_RUN="true"
            shift
            ;;
        -e|--exclude)
            EXCLUDE_FILE="$2"
            shift 2
            ;;
        -l|--log)
            LOG_FILE="$2"
            shift 2
            ;;
        --help)
            show_usage
            exit 0
            ;;
        -*)
            echo "Unknown option: $1"
            show_usage
            exit 1
            ;;
        *)
            if [ -z "$SOURCE_DIR" ]; then
                SOURCE_DIR="$1"
            elif [ -z "$TARGET_DIR" ]; then
                TARGET_DIR="$1"
```

```
            fi
            shift
            ;;
    esac
done

# Validate required arguments
if [ -z "$SOURCE_DIR" ] || [ -z "$TARGET_DIR" ]; then
    echo "Error: Both source and target directories are required"
    show_usage
    exit 1
fi

main
```

System Maintenance Utilities

Log Rotator

```
#!/bin/bash
# log_rotator.sh - Advanced log rotation utility

CONFIG_FILE="${CONFIG_FILE:-/etc/logrotate_custom.conf}"
LOG_DIR="${LOG_DIR:-/var/log}"
MAX_AGE_DAYS="${MAX_AGE_DAYS:-30}"
COMPRESS="${COMPRESS:-true}"
COMPRESSION_DELAY="${COMPRESSION_DELAY:-1}"

# Default configuration
declare -A default_config=(
    [rotate]="7"
    [size]="100M"
    [compress]="true"
    [delaycompress]="true"
    [missingok]="true"
    [notifempty]="true"
    [create]="644 root root"
)

# Logging function
log() {
    echo "$(date '+%Y-%m-%d %H:%M:%S') - $*" | tee -a "/var/log/log_rotator.log"
}
```

```
# Parse size string to bytes
parse_size() {
    local size_str="$1"
    local size_num=$(echo "$size_str" | sed 's/[^0-9]*//g')
    local size_unit=$(echo "$size_str" | sed 's/[0-9]*//g' | tr '[:lower:]' '[:upper:]')

    case "$size_unit" in
        K|KB) echo $((size_num * 1024)) ;;
        M|MB) echo $((size_num * 1024 * 1024)) ;;
        G|GB) echo $((size_num * 1024 * 1024 * 1024)) ;;
        *) echo "$size_num" ;;
    esac
}

# Check if file needs rotation
needs_rotation() {
    local log_file="$1"
    local max_size="$2"
    local rotate_count="$3"

    # Check if file exists and is not empty
    if [ ! -f "$log_file" ] || [ ! -s "$log_file" ]; then
        return 1
    fi

    # Check size
    local file_size=$(stat -c%s "$log_file" 2>/dev/null || stat -f%z "$log_file" 2>/dev/null
    local max_bytes=$(parse_size "$max_size")

    if [ "$file_size" -gt "$max_bytes" ]; then
        return 0
    fi

    # Check if we have reached max rotations
    if [ -f "${log_file}.${rotate_count}" ]; then
        return 0
    fi

    return 1
}

# Rotate a single log file
rotate_log() {
    local log_file="$1"
    local config_name="$2"

    # Load configuration for this log
```

```
    local rotate_count="${log_configs[$config_name.rotate]:-${default_config[rotate]}}"
    local max_size="${log_configs[$config_name.size]:-${default_config[size]}}"
    local compress="${log_configs[$config_name.compress]:-${default_config[compress]}}"
    local delaycompress="${log_configs[$config_name.delaycompress]:-${default_config[delayco
    local missingok="${log_configs[$config_name.missingok]:-${default_config[missingok]}}"
    local notifempty="${log_configs[$config_name.notifempty]:-${default_config[notifempty]}}
    local create_mode="${log_configs[$config_name.create]:-${default_config[create]}}"

    # Check if rotation is needed
    if ! needs_rotation "$log_file" "$max_size" "$rotate_count"; then
        log "No rotation needed for: $log_file"
        return 0
    fi

    log "Rotating log file: $log_file"

    # Remove oldest log if it exists
    if [ -f "${log_file}.${rotate_count}" ]; then
        rm "${log_file}.${rotate_count}"
        log "Removed oldest log: ${log_file}.${rotate_count}"
    fi

    # Rotate existing logs
    for ((i=rotate_count-1; i>=1; i--)); do
        local current_log="${log_file}.${i}"
        local next_log="${log_file}.$((i+1))"

        if [ -f "$current_log" ]; then
            mv "$current_log" "$next_log"
            log "Moved: $current_log -> $next_log"
        fi
    done

    # Move current log to .1
    if [ -f "$log_file" ]; then
        mv "$log_file" "${log_file}.1"
        log "Moved: $log_file -> ${log_file}.1"

        # Create new log file with proper permissions
        if [ -n "$create_mode" ]; then
            local mode=$(echo "$create_mode" | awk '{print $1}')
            local owner=$(echo "$create_mode" | awk '{print $2}')
            local group=$(echo "$create_mode" | awk '{print $3}')

            touch "$log_file"
            chmod "$mode" "$log_file" 2>/dev/null || true
```

```
            chown "$owner:$group" "$log_file" 2>/dev/null || true

            log "Created new log file: $log_file ($mode $owner:$group)"
        fi
    fi

    # Compress rotated logs
    if [ "$compress" = "true" ]; then
        local start_index=1
        if [ "$delaycompress" = "true" ]; then
            start_index=2
        fi

        for ((i=start_index; i<=rotate_count; i++)); do
            local log_to_compress="${log_file}.${i}"
            if [ -f "$log_to_compress" ] && [[ ! "$log_to_compress" =~ \.gz$ ]]; then
                if gzip "$log_to_compress"; then
                    log "Compressed: $log_to_compress"
                else
                    log "Failed to compress: $log_to_compress"
                fi
            fi
        done
    fi

    # Execute post-rotation script if configured
    local postrotate_script="${log_configs[$config_name.postrotate]:-}"
    if [ -n "$postrotate_script" ]; then
        log "Executing post-rotation script: $postrotate_script"
        if eval "$postrotate_script"; then
            log "Post-rotation script completed successfully"
        else
            log "Post-rotation script failed"
        fi
    fi
}

# Load configuration file
load_config() {
    declare -gA log_configs

    if [ ! -f "$CONFIG_FILE" ]; then
        log "Configuration file not found: $CONFIG_FILE"
        return 1
    fi

    local current_section=""
```

```
    while IFS= read -r line; do
        # Skip comments and empty lines
        [[ $line =~ ^[[:space:]]*# ]] && continue
        [[ -z $line ]] && continue

        # Check for section header
        if [[ $line =~ ^\[([^\]]+)\] ]]; then
            current_section="${BASH_REMATCH[1]}"
            continue
        fi

        # Parse key=value pairs
        if [[ $line =~ ^[[:space:]]*([^=]+)=(.*)$ ]]; then
            local key="${BASH_REMATCH[1]// /}"
            local value="${BASH_REMATCH[2]}"

            if [ -n "$current_section" ]; then
                log_configs["$current_section.$key"]="$value"
            else
                log_configs["$key"]="$value"
            fi
        fi
    done < "$CONFIG_FILE"

    log "Configuration loaded from: $CONFIG_FILE"
}

# Generate default configuration
generate_config() {
    cat > "$CONFIG_FILE" << EOF
# Log Rotation Configuration
# Generated on $(date)

# Global settings
compress=true
delaycompress=true
missingok=true
notifempty=true

# Apache logs
[apache]
files=/var/log/apache2/*.log
rotate=52
size=100M
create=644 www-data www-data
postrotate=systemctl reload apache2
```

```
# Nginx logs
[nginx]
files=/var/log/nginx/*.log
rotate=52
size=100M
create=644 www-data www-data
postrotate=systemctl reload nginx

# System logs
[syslog]
files=/var/log/syslog
rotate=7
size=50M
create=644 syslog adm

# Application logs
[application]
files=/var/log/myapp/*.log
rotate=30
size=10M
create=644 myapp myapp
postrotate=killall -USR1 myapp
EOF

    log "Configuration template created: $CONFIG_FILE"
}

# Clean old compressed logs
cleanup_old_logs() {
    log "Cleaning up logs older than $MAX_AGE_DAYS days"

    local deleted_count=0
    local freed_space=0

    find "$LOG_DIR" -name "*.gz" -mtime +$MAX_AGE_DAYS -type f | while read -r old_log; do
        local file_size=$(stat -c%s "$old_log" 2>/dev/null || stat -f%z "$old_log" 2>/dev/nu

        if rm "$old_log"; then
            log "Deleted old log: $old_log ($(numfmt --to=iec-i --suffix=B $file_size))"
            ((deleted_count++))
            freed_space=$((freed_space + file_size))
        fi
    done

    log "Cleanup completed: $deleted_count files deleted, $(numfmt --to=iec-i --suffix=B $fr
}
```

```
# Show log statistics
show_statistics() {
    echo "Log File Statistics"
    echo "==================="
    echo "Generated: $(date)"
    echo

    # Count log files by type
    echo "Log Files by Extension:"
    echo "======================="
    find "$LOG_DIR" -type f -name "*.log*" | sed 's/.*\.//' | sort | uniq -c | sort -nr

    echo
    echo "Largest Log Files:"
    echo "==================="
    find "$LOG_DIR" -type f -name "*.log*" -exec ls -lh {} \; | sort -k5 -hr | head -10

    echo
    echo "Disk Usage by Directory:"
    echo "========================"
    du -h "$LOG_DIR"/* 2>/dev/null | sort -hr | head -10

    echo
    echo "Compressed vs Uncompressed:"
    echo "==========================="
    local compressed_count=$(find "$LOG_DIR" -name "*.gz" -type f | wc -l)
    local uncompressed_count=$(find "$LOG_DIR" -name "*.log" -type f | wc -l)
    local compressed_size=$(find "$LOG_DIR" -name "*.gz" -type f -exec stat -c%s {} \; 2>/de
    local uncompressed_size=$(find "$LOG_DIR" -name "*.log" -type f -exec stat -c%s {} \; 2>

    echo "Compressed files: $compressed_count ($(numfmt --to=iec-i --suffix=B $compressed_si
    echo "Uncompressed files: $uncompressed_count ($(numfmt --to=iec-i --suffix=B $uncompres
}

# Main rotation process
main() {
    log "Starting log rotation process"

    if ! load_config; then
        log "Failed to load configuration, using defaults"
    fi

    # Get list of configured log sections
    local sections=()
    for key in "${!log_configs[@]}"; do
        if [[ $key =~ ^([^.]+)\.files$ ]]; then
```

```
            sections+=("${BASH_REMATCH[1]}")
        fi
    done

    # Remove duplicates
    IFS=$'\n' sections=($(sort -u <<<"${sections[*]}"))
    unset IFS

    if [ ${#sections[@]} -eq 0 ]; then
        log "No log sections configured, rotating all .log files in $LOG_DIR"

        # Rotate all .log files with default settings
        find "$LOG_DIR" -name "*.log" -type f | while read -r log_file; do
            rotate_log "$log_file" "default"
        done
    else
        # Rotate configured logs
        for section in "${sections[@]}"; do
            local files_pattern="${log_configs[$section.files]}"

            if [ -n "$files_pattern" ]; then
                log "Processing section: $section ($files_pattern)"

                # Expand file pattern
                for log_file in $files_pattern; do
                    if [ -f "$log_file" ]; then
                        rotate_log "$log_file" "$section"
                    fi
                done
            fi
        done
    fi

    # Cleanup old logs
    cleanup_old_logs

    log "Log rotation process completed"
}

# Show usage
show_usage() {
    cat << EOF
Log Rotator - Advanced log rotation utility

Usage: $0 [command] [options]

Commands:
```

```
  rotate                  Perform log rotation (default)
  config                  Generate configuration template
  stats                   Show log file statistics
  cleanup                 Clean up old compressed logs only

Options:
  -c, --config FILE       Configuration file path
  -d, --log-dir DIR       Log directory path
  -a, --max-age DAYS      Maximum age for log cleanup
  --help                  Show this help

Environment Variables:
  CONFIG_FILE             Configuration file path
  LOG_DIR                Log directory path
  MAX_AGE_DAYS           Maximum age for cleanup

Examples:
  $0                      # Rotate logs using default config
  $0 config               # Generate configuration template
  $0 stats                # Show log statistics
  $0 cleanup --max-age 7 # Clean logs older than 7 days
EOF
}

# Parse command line arguments
COMMAND="rotate"

while [[ $# -gt 0 ]]; do
    case $1 in
        rotate|config|stats|cleanup)
            COMMAND="$1"
            shift
            ;;
        -c|--config)
            CONFIG_FILE="$2"
            shift 2
            ;;
        -d|--log-dir)
            LOG_DIR="$2"
            shift 2
            ;;
        -a|--max-age)
            MAX_AGE_DAYS="$2"
            shift 2
            ;;
        --help)
```

```
            show_usage
            exit 0
            ;;
        -*)
            echo "Unknown option: $1"
            show_usage
            exit 1
            ;;
        *)
            shift
            ;;
    esac
done

# Execute command
case "$COMMAND" in
    rotate)
        main
        ;;
    config)
        generate_config
        ;;
    stats)
        show_statistics
        ;;
    cleanup)
        cleanup_old_logs
        ;;
    *)
        echo "Unknown command: $COMMAND"
        show_usage
        exit 1
        ;;
esac
```

These utility scripts provide comprehensive solutions for common file management and system maintenance tasks. Each script includes extensive configuration options, error handling, and detailed logging to make them suitable for production use.