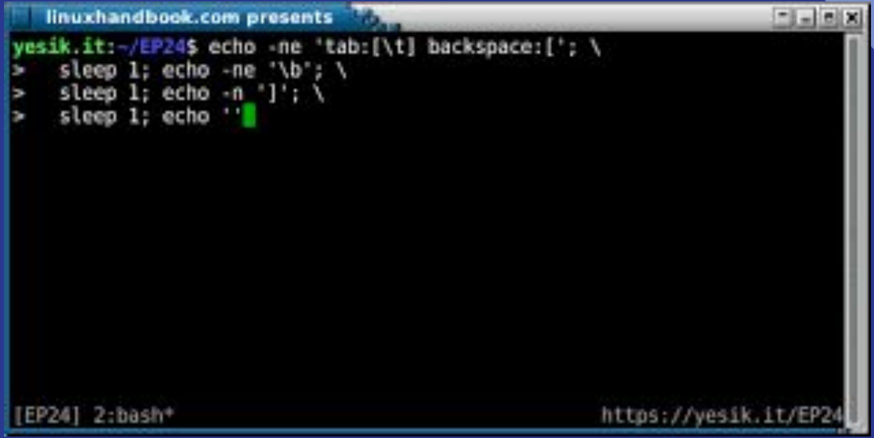
Welcome to Shell Scripting!



```
yesik.it:~/EP24$ echo -ne 'tab:[\t] backspace:['; \
>   sleep 1; echo -ne '\b'; \
>   sleep 1; echo -n ']'; \
>   sleep 1; echo ''
```

```
[EP24] 2:bash*                                      https://yesik.it/EP24
```

Mostafa Al Mahmud

Sr. Software Engineer
AWS Community Builder
https://www.linkedin.com/in/md-mostafa/

Worked in the projects of some valuable brands

SUZUKI
BRAIN STATION 23
HandyMama
COGWHEEL MARKETING
CONF42
ROGERS
Build With Innovation
EVOLVE. GROW. THRIVE.
Synspective
incepta
RAZIB MARKETING
COGWHEEL ANALYTICS
rehive

Community Activities

aws
How to Achieve Reliability on AWS Well-architected Framework
Mostafa Al Mahmud
AWS Certified WAFR, Brain Station 23 Ltd.
BRAIN STATION
aws community voices
Join Online
WEBINAR
HOW TO ACHIEVE RELIABILITY WITH AN AWS WELL-ARCHITECTED FRAMEWORK
12.00PM-1.00PM
20TH OCTOBER
Speaker
MOSTAFA AL MAHMUD
SOFTWARE ENGINEER & CERTIFIED WAFR REVIEWER
Organized By
BRAIN STATION 23
AWS community builders
SPEAKER
daffodil
AWS USER GROUP BANGLADESH
BASIS
Build Global Scale Cloud Native SaaS Application
CONF42 CLOUD NATIVE 2024
MARCH 21 • ONLINE
Step Functions
Execution: 835acf96-4fdf-4069-88ee-40c66f54a64a
aws
COMMUNITY DAY
BANGLADESH
mpergola@amazon.com (Host)
Lily - Community Man...
Pradhyumna N Holla

©Md Mostafa Al Mahmud

Learning Path Overview

- History and Importance of Shell Scripting in DevOps
- Types of Shells (Bash, ZSH, etc.)
- Overview of DevOps Automation
- Basic Syntax and Structure of a Shell Script
- Variables, Loops, Conditionals, and others in Shell Scripting
- Practical: Hello World! Script, Variable Declaration and Usage, and Basic Input/Output

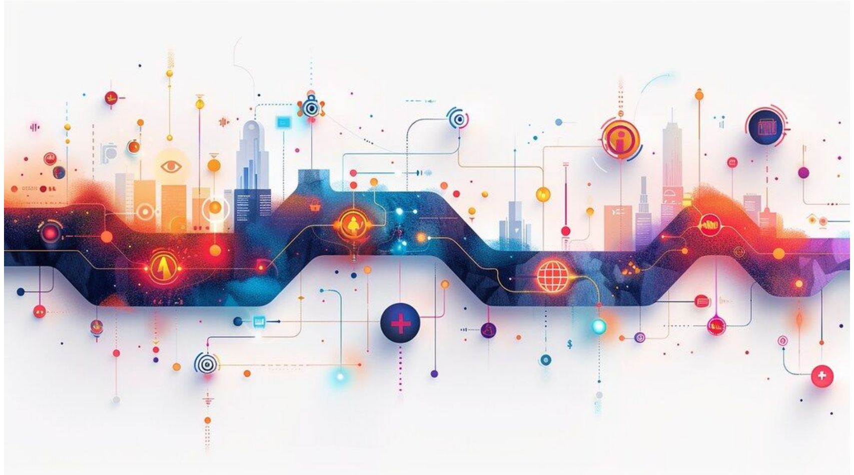
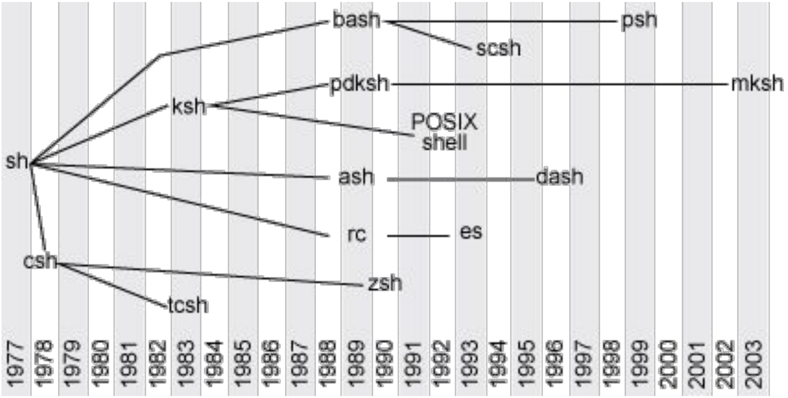


Image credit: https://flickr.com

History of Shell Scripting (Early Developments)

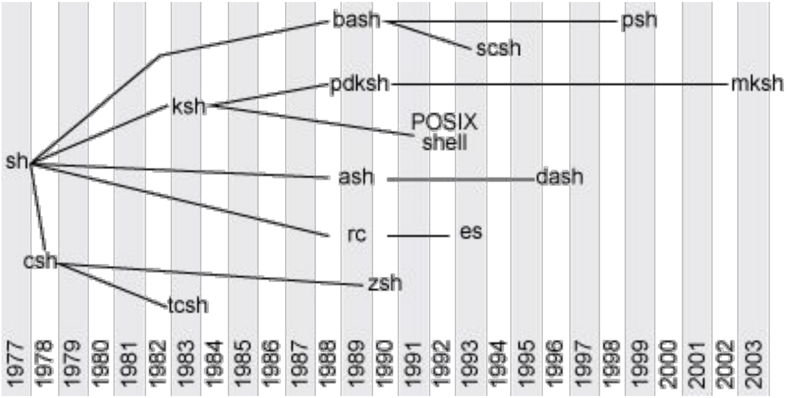
- **1971: Thompson Shell**
  - ➢ First UNIX shell, basic command interpreter.
  - ➢ Limited functionality, lacked scripting features.
- **1977: Bourne Shell (sh)**
  - ➢ Created by Stephen Bourne at AT&T Bell Labs.
  - ➢ Introduced scripting capabilities (control flows, loops, variables).
  - ➢ Served as both interactive command interpreter and scripting tool.



Source: https://developer.ibm.com/tutorials/l-linux-shells/

Evolution of Modern Shells

- **Bourne Shell Advancements**
  - Command substitution and HERE documents.
  - Integrated signal handling, no function definitions.
- **Derivative Shells**
  - **Korn Shell (ksh)**: Enhanced scripting features.
  - **Almquist Shell (ash)**: Lightweight, used in embedded systems.
  - **Bourne Again Shell (Bash)**: Most widely used today in Linux.
  - **C Shell (csh)**: Developed around the same time, introduced new syntax (e.g., C-like syntax).
- **Impact on DevOps**
  - Foundation for automation and configuration management.
  - Essential tool in DevOps pipelines for scripting and automation.



Source: https://developer.ibm.com/tutorials/l-linux-shells/

Types of Shell Scripting

- ❖ **Bourne Shell (sh)**
  - ➢ Basic syntax
  - ➢ Original UNIX shell, known for simplicity
  - ➢ Basis for many modern shells
  - ➢ Minimal features, foundational
  - ➢ Path Name: /bin/sh or /sbin/sh
  - ➢ Prompt for the root user: #
  - ➢ Prompt for the non-root user: $

```
/bin/sh or /sbin/sh

echo "Hello, World!"  # Simple command
if [ condition ]; then  # Conditional
    commands
Fi

#!/bin/sh
if [ $days -gt 365 ]
then
   echo This is over a year.
fi
```

Types of Shell Scripting

- ❖ **Bash (Bourne-Again Shell)**
  - ➢ A script is a sequence of commands saved in a file, allowing you to automate tasks and streamline your workflow.
  - ➢ A powerful command-line interpreter.
  - ➢ Most widely used, default in many Linux distributions.
  - ➢ Enhanced features like command history, job control.
  - ➢ Path Name: /bin/bash

Image credit: https://bashlogo.com

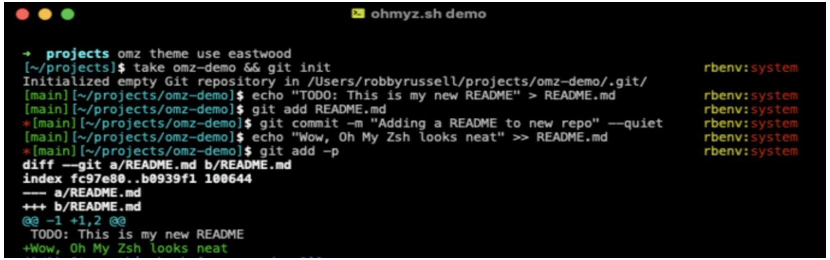
Types of Shell Scripting

- **C Shell (csh)**
  - C-like syntax, useful for users familiar with C programming.
  - Features include aliasing and scripting capabilities.
  - Built-in math operations
  - Path Name: /bin/csh
  - Prompt for the root user: hostname#
  - Prompt for the non-root user: hostname%

```
#!/bin/csh

echo "Hello, World!"  # Simple
command
if (condition) then  # Conditional
    commands
endif

if ( $days > 365 ) then
   echo This is over a year.
endif
```

Types of Shell Scripting

- **Korn Shell (ksh)**
  - Combines features of Bourne and C shells.
  - Enhanced scripting and interactive use.
  - Path Name: /bin/ksh
  - Prompt for the root user: #
  - Prompt for the non-root user: $
- **Z Shell (zsh)**
  - Combines features of Bash, ksh, and csh.
  - Highly customizable, popular for interactive use.



```
→  projects omz theme use eastwood
[~/projects]$ take omz-demo && git init                                    rbenv:system
Initialized empty Git repository in /Users/robbyrussell/projects/omz-demo/.git/
[main][~/projects/omz-demo]$ echo "TODO: This is my new README" > README.md  rbenv:system
[main][~/projects/omz-demo]$ git add README.md                               rbenv:system
*[main][~/projects/omz-demo]$ git commit -m "Adding a README to new repo" --quiet  rbenv:system
[main][~/projects/omz-demo]$ echo "Wow, Oh My Zsh looks neat" >> README.md   rbenv:system
*[main][~/projects/omz-demo]$ git add -p                                     rbenv:system
diff --git a/README.md b/README.md
index fc97e80..b0939f1 100644
--- a/README.md
+++ b/README.md
@@ -1 +1,2 @@
 TODO: This is my new README
+Wow, Oh My Zsh looks neat
```

```
echo "Hello, World!"  # Simple command
if [[ condition ]]; then  # Conditional
    commands
Fi

#!/bin/ksh

if [[ "$status" -eq 0 ]]; then
    echo "Server update successfully and
restarting…"
    # restart command
else
    echo "Something error. Try again"
fi
```

Importance of Shell Scripting in DevOps

- Automation
- Portability
- Accessibility
- Flexibility
- Integration
- Debugging
- Simple and Efficient

Image credit: Etienne Girardet on Unsplash

Why Learn Shell Scripting?

- Automate repetitive tasks - Save time and effort
- Manage infrastructure efficiently
- Control servers and applications
- Improve consistency and reliability
- Reduce human error
- Enhance your DevOps skills set
- Become a valuable asset

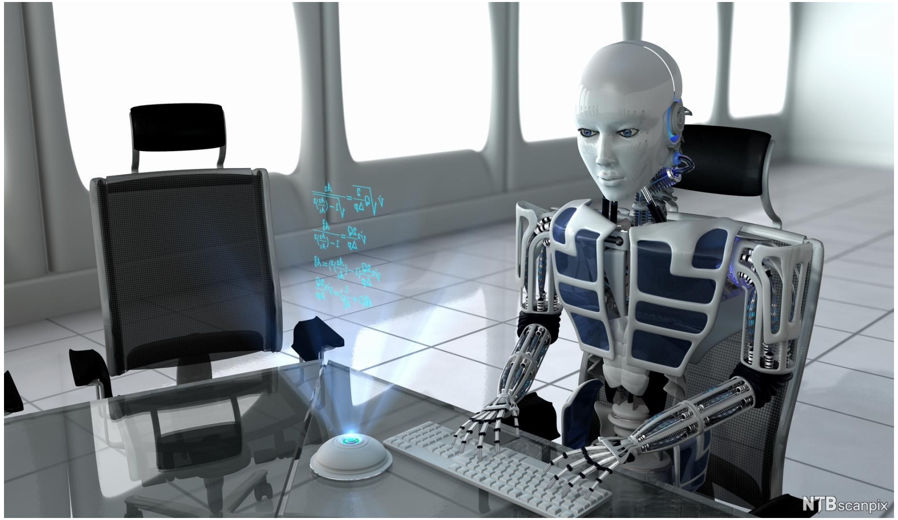


Image credit: https://ndla.no/

Basic Bash Concepts and Features

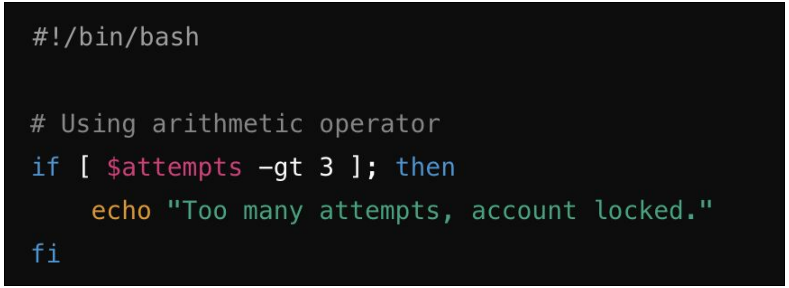
- **Variables - Store and manipulate data.**

```
#!/bin/bash

# Declare a variable
current_user=$(whoami)
echo "Current user: $current_user"
```

Basic Bash Concepts and Features

- **Operators - Perform calculations and comparisons.**

```
#!/bin/bash

# Using arithmetic operator
if [ $attempts -gt 3 ]; then
    echo "Too many attempts, account locked."
fi
```

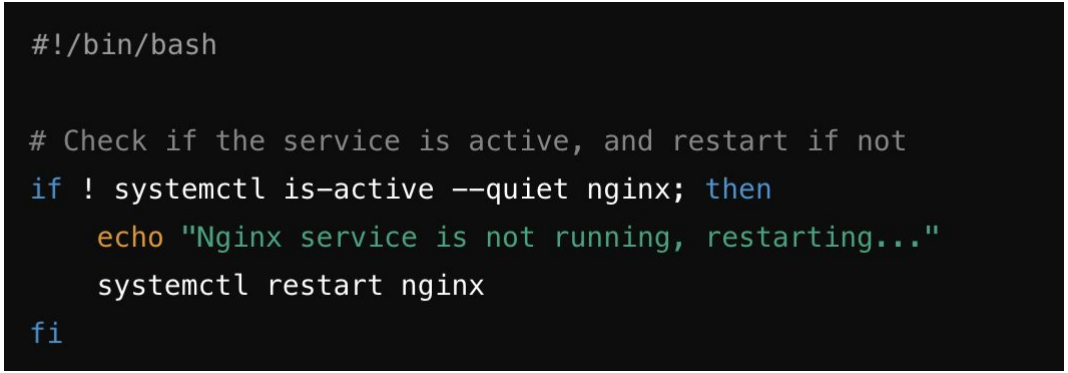


```
#!/bin/bash

# Using arithmetic operator
if [ $attempts -gt 3 ]; then
    echo "Too many attempts, account locked."
fi
```

Basic Bash Concepts and Features

- **Conditional statements - Make decisions based on conditions.**

```
#!/bin/bash

# Check if the service is active, and restart if not
if ! systemctl is-active --quiet nginx; then
    echo "Nginx service is not running, restarting..."
    systemctl restart nginx
fi
```

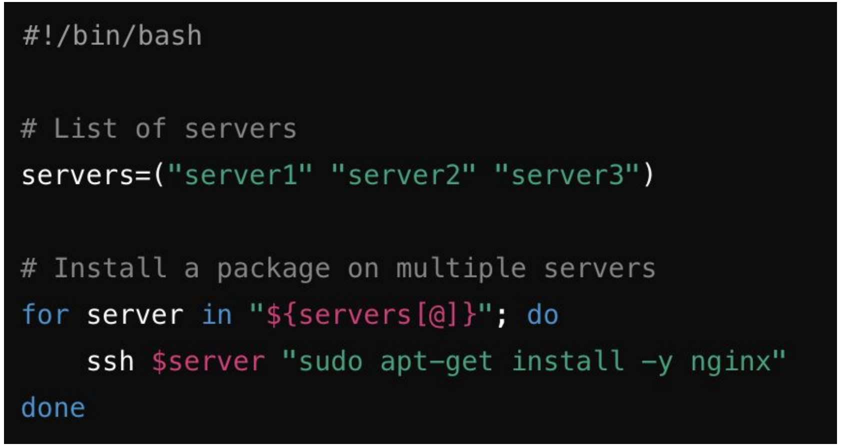


```
#!/bin/bash

# Check if the service is active, and restart if not
if ! systemctl is-active --quiet nginx; then
    echo "Nginx service is not running, restarting..."
    systemctl restart nginx
fi
```

Basic Bash Concepts and Features

- **Loops - Repeat actions multiple times.**

```
#!/bin/bash

# List of servers
servers=("server1" "server2" "server3")

# Install a package on multiple servers
for server in "${servers[@]}"; do
    ssh $server "sudo apt-get install -y nginx"
done
```



```
#!/bin/bash

# List of servers
servers=("server1" "server2" "server3")

# Install a package on multiple servers
for server in "${servers[@]}"; do
    ssh $server "sudo apt-get install -y nginx"
done
```

Basic Bash Concepts and Features

- **Functions - Reusable blocks of code.**

```
#!/bin/bash

# Function to install package on a server
install_package() {
    server=$1
    package=$2
    ssh $server "sudo apt-get install -y $package"
}

# Call the function for multiple servers
install_package "server1" "nginx"
install_package "server2" "apache2"
```

Basic Bash Concepts and Features

- **Command Execution - Invoke system utilities, applications, and others**

```
# Check disk space usage

df -h
```

- **Redirection and Pipelines - I/O streams and chain commands together**

```
#!/bin/bash

# Redirect output and use pipeline to filter errors

cat /var/log/syslog | grep "error" > error_log.txt
```

- **And many more...**

Getting Started with Bash

- Terminal basics - Navigation, commands, and shortcuts
- Running commands - Executing basic commands in the terminal
- Command line interface (CLI) - Understanding user interaction
- Exit status and return codes - Interpreting script execution results

```
#exit example…

# Check if a file exists
if [ -f "/path/to/file" ]; then
    echo "File exists."
else
    echo "File does not exist."
    exit 1  # Non-zero exit status indicates an error
fi
```

Getting Started with Bash

Terminal basics - Navigation, commands, and shortcuts.

Navigation:

- Up/Down arrows: Cycle through previous commands.
- Tab: Autocomplete commands and file names.
- Ctrl+L: Clear the screen.
- Ctrl+A/Alt+B: Move cursor to beginning/end of line.
- Ctrl+D: Exit the terminal.
- ls: List files and directories in the current directory.
- cd: Change directory (e.g., cd documents).
- pwd: Print the current working directory.
- man: Get help on a specific command (e.g., man ls).

Shortcuts:

- Ctrl+C: Cancel a running command.
- Ctrl+Z: Suspend a running command (bring back with fg).
- Ctrl+R: Search command history.
- Ctrl+S/X: Pause/Resume output scrolling.

Getting Started with Bash

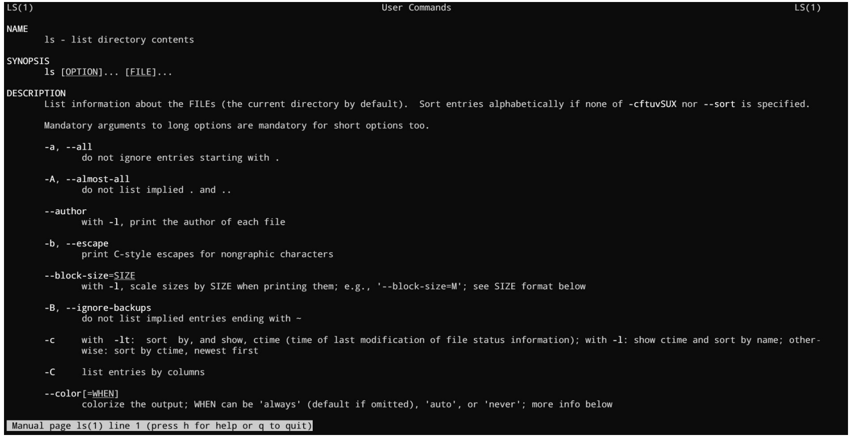
Running commands - Executing basic commands in the terminal

- Commands are instructions executed by the terminal.
- Enter a command, press Enter, and observe the output.
- Combine commands with options and arguments for specific actions.
- Examples:
- ls -l (list files with details).
- cd .. (move to the parent directory).
- man cp (get help on the copy command).
- mkdir documents (create a new directory).
- touch test.txt (create an empty file).
- cat test.txt (display the contents of a file).
- cp file1 file2 (copy a file).
- rm file.txt (remove a file).

Getting Started with Bash

Command line interface (CLI) - Understanding user interaction

- CLI: Text-based interface for interacting with the system.
- User types commands, system responds with output.
- Understand prompts and messages carefully.
- Provide necessary input when prompted.
- Experiment with interactive commands (e.g., cal for calendar).



```
LS(1)                                   User Commands                                   LS(1)

NAME
       ls - list directory contents

SYNOPSIS
       ls [OPTION]... [FILE]...

DESCRIPTION
       List information about the FILEs (the current directory by default).  Sort entries alphabetically if none of -cftuvSUX nor --sort is specified.

       Mandatory arguments to long options are mandatory for short options too.

       -a, --all
              do not ignore entries starting with .

       -A, --almost-all
              do not list implied . and ..

       --author
              with -l, print the author of each file

       -b, --escape
              print C-style escapes for nongraphic characters

       --block-size=SIZE
              with -l, scale sizes by SIZE when printing them; e.g., '--block-size=M'; see SIZE format below

       -B, --ignore-backups
              do not list implied entries ending with ~

       -c     with  -lt:  sort  by, and show, ctime (time of last modification of file status information); with -l: show ctime and sort by name; other-
              wise: sort by ctime, newest first

       -C     list entries by columns

       --color[=WHEN]
              colorize the output; WHEN can be 'always' (default if omitted), 'auto', or 'never'; more info below

 Manual page ls(1) line 1 (press h for help or q to quit)
```

Image credit: freecodecamp

Writing your first "Hello World!" script

- Open a text editor (e.g., nano, Vim) and create a new file
- Name the file with a .sh extension (e.g., hello.sh)
- Type the following line:

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

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

Variables

- **Storage location that have a name**
- **Name-value pairs**
- **Case-sensitive**
- **UPPERCASE followed by convention**

Syntex:
VARIABLE_NAME=”value”

```
#!/bin/bash

MY_SHELL="bash"

echo "I like the $MY_SHELL shell."
```

```
MY_SHELL="bash"

echo "I like the ${MY_SHELL}

shell."
```

Variables - examples

```
#!/bin/bash
MY_SHELL="bash"
echo "I am ${MY_SHELL}ing on my keyboard."
```

Output:

```
I am bashing on my keyboard.
```

```
MY_SHELL="bash"
echo "I am $MY_SHELLing on my keyboard."
```

```
I am  on my keyboard.
```

Variables - examples

```
#!/bin/bash
SERVER_NAME=$(hostname)
echo "You are running this script on ${SERVER_NAME}."
```

Output:

```
You are running this script on
{servername}
```

Using backtick `

```
SERVER_NAME=`hostname`
echo "You are running this script on ${SERVER_NAME}."
```

```
You are running this script on
{servername}
```

Variables

Two types of variables

1. System-defined variables

- SHELL
- PWD
- LOGNAME
- HOME
- SSH_CONNECTION
- SSH_CLIENT
- USER
- PATH
- and so on

2. User-defined variables

- VAR_1=”value1”
- HOST_NAME=$(hostname)
- Create as needed

Variables Names - Valid vs Invalid

Invalid

- **123var** (starts with a number)
- **var-name** (contains a hyphen)
- **my variable** (contains whitespace)
- **!special** (contains special characters other than underscores)
- **var#1, va^riable, @variable** (contains special characters other than underscores)
- **my.variable** (contains special characters other than underscores)
- **var?name** (contains special characters other than underscores)
- **var with spaces** (contains whitespace)
- **function** (reserved keyword in Bash)

Valid

- var
- my_variable
- _underscore
- var123
- MY_VARIABLE
- array[0]
- var_name_with_underscores
- VAR_1
- var_with_123_numbers
- PATH

Special Variables

- **$@** - Stores arguments as an array
- **$#** - Show the number of arguments supplied in a given script
- **$$** - Displays the process ID of the current shell
- **$*** - Groups all given arguments by connecting them together
- **$!** - Shows the ID of the last background job
- **$?** - Displays the exit status code for the latest executed command
- **$0** - Displays the filename of the current script
- **$_** - Sets the variable to the latest argument of the last command
- **$-** - Displays the currently used flags on bash shell
- **$1-${11}** - Store data of the first 11 argument names

Special Variables - examples

```
# $@ - Stores arguments as an array
echo "Arguments stored in \$@:"
for arg in "$@"; do
    echo "$arg"
done

# $# - Show the number of arguments supplied in a
given script
echo "Number of arguments: $#"

# $$ - Displays the process ID of the current shell
echo "Process ID of the current shell: $$"
```

```
# $* - Groups all given arguments by
connecting them together
echo "Arguments grouped together using \$*:"
echo "$*"

# $! - Shows the ID of the last background job
echo "ID of the last background job: $!"

# $? - Displays the exit status code for the
latest executed command
echo "Exit status code of the previous
command: $?"
echo
```

File Operators

File operators are used to check various attributes and properties of files.

- **-e** FILE: True if FILE exists.
- **-f** FILE: True if FILE exists and is a regular file.
- -d FILE: True if FILE exists and is a directory.
- -r FILE: True if FILE exists and is readable.
- -w FILE: True if FILE exists and is writable.
- -x FILE: True if FILE exists and is executable.
- -s FILE: True if FILE exists and has a size greater than zero.
- -L FILE: True if FILE exists and is a symbolic link.
- -G FILE: True if FILE exists and is owned by the effective group ID.
- -O FILE: True if FILE exists and is owned by the effective user ID.

File Operators - Examples

```
if [ -e file.txt ]; then
    echo "File exists"
fi

if [ -f file.txt ]; then
    echo "File is a regular file"
fi

if [ -d directory ]; then
    echo "Directory exists"
fi

if [ -r file.txt ]; then
    echo "File is readable"
fi

if [ -s file.txt ]; then
    echo "File is not empty"
fi
```

```
if [ -w file.txt ]; then
    echo "File is writable"
fi

if [ -x script.sh ]; then
    echo "Script is executable"
fi

if [ -x script.sh ]; then
    echo "Script is executable"
fi

if [ -x script.sh ]; then
    echo "Script is executable"
fi
```

String Operators

String operators are used to manipulate and compare strings

- **-z:** This operator returns true if the length of the string is zero (i.e., the string is empty).
- **-n:** This operator returns true if the length of the string is non-zero (i.e., the string is not empty).
- **Length Operator - ${#string}:** Returns the length of the string.

- **Substring Removal (Prefix):**
  - **${string#substring}:** Removes the shortest match of substring from the beginning of the string.
  - **${string##substring}:** Removes the longest match of substring from the beginning of the string.
- **Substring Removal (Suffix):**
  - **${string%substring}:** Removes the shortest match of substring from the end of the string.
  - **${string%%substring}:** Removes the longest match of substring from the end of the string.

String Operators

- **Substring Extraction - ${string:start:length}:** Extracts a substring starting at the specified position with the specified length.

- **Substring Replacement:**
  - **${string/substring/replacement}:** Replaces the first occurrence of substring with replacement.
  - **${string//substring/replacement}:** Replaces all occurrences of substring with replacement.

- **Substring Test:${string:substring}:** Tests if substring is present in string.
  If present, returns true (0); otherwise, returns false (1).

String Operators - Examples

```
string="Hello, World!"
echo "Length of the string: ${#string}"
# Output: 13

echo "Substring: ${string:7:5}"
# Output: World

echo "Prefix Removal: ${string#Hello, }"
# Output: World!

echo "Suffix Removal: ${string%World!}"
# Output: Hello,

echo "Substring Replacement: ${string/Hello/Hi}"
# Output: Hi, World!
```

```
if [[ $string == *"Hello"* ]]; then
    echo "Substring 'Hello' is present."
else
    echo "Substring 'Hello' is not present."
fi

# Output: Substring 'Hello' is present.
```

String Operators - Example (-z)

```
#!/bin/bash

string1="Hello"
string2=""

if [ -z "$string1" ]; then
    echo "string1 is empty."
else
    echo "string1 is not empty."
fi

if [ -z "$string2" ]; then
    echo "string2 is empty."
else
    echo "string2 is not empty."
fi
```

Output:

```
string1 is not empty.



string2 is empty.
```

String Operators - Example (-n)

```
#!/bin/bash

string1="Hello"
string2=""

if [ -n "$string1" ]; then
    echo "string1 is not empty."
else
    echo "string1 is empty."
fi

if [ -n "$string2" ]; then
    echo "string2 is not empty."
else
    echo "string2 is empty."
fi
```

Output:

```
string1 is not empty.

string2 is empty.
```

Arithmetic Operators

Arithmetic operators are used to perform mathematical operations on numeric values.

Addition (+): Adds two numbers.

```
sum=$((5 + 3))
echo "Sum: $sum" # Output: 8
```

Subtraction (-): Subtracts the second number from the first.

```
difference=$((10 - 3))
echo "Difference: $difference" # Output: 7
```

Multiplication (*): Multiplies two numbers.

```
product=$((5 * 4))
echo "Product: $product" # Output: 20
```

Division (/): Divides the first number by the second. Note: If the divisor is zero, Bash will throw an error.

```
quotient=$((20 / 5))
echo "Quotient: $quotient" # Output: 4
```

Arithmetic Operators

Modulus (%): Returns the remainder of the division operation.

```
remainder=$((10 % 3))
echo "Remainder: $remainder" # Output: 1
```

Increment (++): Increases the value of a variable by 1.

```
count=5
((count++))
echo "Incremented value: $count" # Output: 6
```

Decrement (--): Decreases the value of a variable by 1.

```
count=5
((count--))
echo "Decremented value: $count" # Output: 4
```

Comparison Operators with Arithmetic Value

Equal (-eq): Checks if two values are equal.

```
if [ "$num1" -eq "$num2" ]; then
    echo "num1 is equal to num2"
fi
```

Not Equal (-ne): Checks if two values are not equal.

```
if [ "$num1" -ne "$num2" ]; then
    echo "num1 is not equal to num2"
fi
```

Greater Than (-gt): Checks if the first value is greater than the second.

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

Greater Than or Equal To (-ge): Checks if the first value is greater than or equal to the second.

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

num1=5
num2=10

Comparison Operators with Arithmetic Value

Less Than (-lt): Checks if the first value is less than the second.

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

Less Than or Equal To (-le): Checks if the first value is less than or equal to the second.

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

```
#!/bin/bash

num1=5
num2=10

if [ "$num1" -eq "$num2" ]; then
    echo "num1 is equal to num2"
elif [ "$num1" -lt "$num2" ]; then
    echo "num1 is less than num2"
else
    echo "num1 is greater than num2"
fi
```

Exit Status and Return Codes

- Every command returns an exit status
- Exit status: Numerical code indicating script execution outcome
- Range from 0-255
- 0: Script executed successfully
- Non-zero: Script encountered errors or issues
- Specific codes may indicate specific errors (e.g., 1: permission denied, 2: no such file or directory, etc.)
- Use echo $? after script execution to check status
- Use for error checking
- Use man or info to find meaning of exit status

Checking the Exit Status

```
ls /not/there/

echo “$?”

Output: 2
```

```
Echo “Hello World”

echo “$?”

Output: 0
```

```
HOST="google.com"

# Check if the host is reachable
ping -c 1 $HOST

# Check the exit status of the ping
command
if [ “$?” -eq “0” ];
then
    echo "$HOST is reachable."
else
    echo "Unable to reach $HOST."
fi
```

Checking the Exit Status

```
HOST="google.com"

# Check if the host is reachable
ping -c 1 $HOST

# Check the exit status of the ping
command
if [ “$?” -ne “0” ];
then
    echo "$HOST is unreachable."
fi
```

```
HOST="google.com"

# Check if the host is reachable
ping -c 1 $HOST
RETURN_CODE=$?

# Check the exit status of the ping
command
if [ “RETURN_CODE” -eq “0” ];
then
    echo "$HOST is unreachable."
fi
```

Checking the Exit Status

&& = AND

```
mkdir /tmp/bak && cp test.txt /tmp/bak/
```

```
#!/bin/bash
HOST="google.com"

# Check if the host is reachable
ping -c 1 $HOST && echo "$HOST is reachable."
```

|| = OR

```
cp test.text /tmp/bak/ || cp test.txt /tmp
```

```
#!/bin/bash
HOST="google.com"

# Check if the host is reachable
ping -c 1 $HOST || echo "$HOST is unreachable."
```

The Semicolon

To get all executed in a single line use a semicolon separating commands

```
# Declare and assign values to variables
NAME="John"; AGE=30; CITY="New York"

# Print the values of variables
echo "Name: $NAME; Age: $AGE; City: $CITY"
```

Same as:

```
NAME="John"
AGE=30
CITY="New York"

echo "Name: $NAME"
echo "Age: $AGE"
echo "City: $CITY"
```

The Semicolon

To get all executed in a single line use a semicolon separating commands

```
cp text.txt /temp/test; cp text.txt /temp
```

Same as:

```
cp text.txt /temp/test

cp text.txt /temp
```

```
ls /not/there; hostname
```

Output:

```
ls: /not/there: No such file or directory
username
```

```
date; uptime
```

Output:

```
Wed Feb 21 15:13:10 +06 2024
15:13  up 13 days, 20:48, 4 users, load averages: 3.07 3.89
4.22
```

The Exit Command

- **Explicitly define the return code**
  - exit 0
  - exit 1
  - exit 2
  - exit 3
  - exit 255
  - etc.
- **The last executed command is the default value**

The Exit Command

```
#!/bin/bash

# Check if a file exists
if [[ ! -f "important_data.txt" ]];
then
  echo "Error: File not found!"
  exit 2
fi

# Process the file successfully
# ...

# All good, signal success!
exit 0
```

```
#!/bin/bash
HOST="google.com"

# Check if the host is reachable
ping -c 1 $HOST

# Check the exit status of the ping
command
if [ “$?” -ne “0” ];
then
    echo "$HOST is unreachable."
    exit 1
fi
exit 0
```

The Exit Command - Summary

- All command return an exit status
- 0 - 255 status codes
- 0 - Success status
- Other than 0 - Error status
- $? Contains the last exit status
- exit command
- Decision making with if, &&, ||

Loops

- **Reading files line-by-line**
- **for loop**
  - Iterates over a list of items
- **while loop**
  - Continues executing a block of code as long as a condition is true
  - Infinite loops
- **until loop**
  - Continues executing a block of code until a condition becomes true
- **\`break\`** and **\`continue\`**
- ***If the command fails, it returns non-zero exit status and then continue the loop until it is closed by force***

```
while [ is_condition_true ];
do
    command 1
    command 2
    …command N
done
```

Loops

While loop

```
while [ is_condition_true ];
do
    command 1
    command 2
    …command N
done
```

# Commands change the condition

Infinite loop

```
while true;
do
    command 1
    command 2
    …command N
done
```

# Commands do not change the condition

Loops (Examples - while, infinite)

```
# Example of a while loop

count=1

while [[ $count -le 5 ]]; do
    echo "Count: $count"
    ((count++))
done
```

The while loop iterates as long as the variable count is less than or equal to 5.

```
# Example of an infinite loop

while true; do
    echo "This loop runs forever!"
done
```

The infinite loop, if uncommented, will continue to execute indefinitely, printing "This loop runs forever!" repeatedly until manually stopped.

Loops (Examples - while, for)

```
# Print numbers from 1 to 5

i=1
while ((i<=5)); do
    echo $i
    ((i++))
done

# Read user input until 'quit' is entered
input=""

while [[ "$input" != "quit" ]]; do
    read -p "Enter something (or 'quit' to exit): " input
    echo "You entered: $input"
done
```

```
# Print numbers from 1 to 5

for ((i=1; i<=5; i++)); do
    echo $i
done

# Iterate over elements in an array

fruits=("apple" "banana" "orange")
for fruit in "${fruits[@]}"; do
    echo "I like $fruit"
done
```

Loops (Examples - until vs while)

```
# Print numbers from 1 to 5

i=1
until ((i>5)); do
    echo $i
    ((i++))
done
```

```
# Keep prompting until 'yes' is entered

input=""
until [[ "$input" == "yes" ]]; do
    read -p "Do you want to continue? (yes/no): "
input
    echo "You entered: $input"
done
```

```
# Example of while loop

counter=0
while (( counter < 5 )); do
    echo "Counter: $counter"
    (( counter++ ))
done
```

# The loop continues executing as long as the condition evaluates to false.

# Once the condition becomes true, the loop terminates.

Loops (Examples - until vs while)

```
# Example: Waiting for a service to become available

echo "Waiting for database service to become available..."

until nc -z localhost 5432; do
    echo "Database service is not yet available. Retrying in 5 seconds..."
    sleep 5
done

echo "Database service is now available. Proceeding with the script."
```

# The until loop continues executing as long as the condition evaluates to true.

# Once the until loop condition becomes false, the loop terminates.

In the provided example, \`nc\` is used to check if a TCP connection to a specific port is successful. The \`-z\` option tells nc to scan for open ports, and if successful, it returns zero indicating the port is open, otherwise, it returns a non-zero exit status indicating the port is closed.

Loops (Examples - until vs while)

```
# Example: Polling a remote server until it responds

echo "Polling remote server..."

# Using 'while' loop to repeatedly send ping requests until a response is received

while ! ping -c 1 -W 1 example.com &> /dev/null; do
    echo "Remote server is not reachable. Retrying in 5 seconds..."
    sleep 5
done

echo "Remote server is reachable. Proceeding with operations."
```

```
# Example: Waiting for a file to be created

echo "Waiting for log file to be created..."

# Using 'until' loop to wait until the file exists

until [[ -f /var/log/application.log ]]; do
    echo "Log file does not exist yet. Retrying in 3 seconds..."
    sleep 3
done

echo "Log file detected. Proceeding with processing."
```

Loops (Examples - until vs while)

- The **until** loop waits until the log file /var/log/application.log is created. It checks for the existence of the file using the -f test condition. If the file does not exist, it prints a message and retries after 3 seconds until the file is created.

- The **while** loop continuously sends ping requests to the remote server example.com until it receives a response. It uses the ping command to send a single ICMP echo request with a timeout of 1 second (-c 1 -W 1). If the server is unreachable, it prints a message and retries after 5 seconds until a response is received.

- **-c** for count and **-w** for deadline in bash ping command
- **!** for negation of the result of the ping command
- **&>** for the combination of both stdout and stdin operators `>` and `2>&1`
- **>** Redirects standard output (stdout) of a command to a file.
- **2>&1** Redirects standard error (stderr) to the same location as stdout.

Loops (Examples - infinite)

```
while true
do
  read -p "Select your choice: 1: uptime.  2: disk usage. " MY_CHOICE
  case "$MY_CHOICE" in
    1)
      uptime
      ;;
    2)
      df -h
      ;;
    *)
      break
      ;;
  esac
done
```

Infinite loop: It continues until it matches `*` and `*break*` statement

Task List

1. Store the output of the command "hostname" in a variable. Display "This script is running on _______." where "_______" is the output of the "hostname" command.
   Hint: It's a best practice to use the ${VARIABLE} syntax if there is text or characters that directly precede or follow the variable.
2. Check exit status with;
   ping -c 1 google.com
   Ping -c 1 -w 1 amazon.com
   Ping -c 1 amazon.com.bangladesh
3. Explore using || (OR) and && (AND) operators with exit codes for chained actions.
4. Write a shell script that displays "man", "bear", "pig", "dog", "cat", and sheep to the screen with each appearing on a separate line. Try to do this in as few lines as possible.
   Hint: Loops can be used to perform repetitive tasks.

https://www.gnu.org/software/bash/manual/html_node/index.html#SEC_Contents

Resource

- https://www.gnu.org/software/bash/manual/html_node/index.html#SEC_Contents
- https://developer.ibm.com/tutorials/l-linux-shells/

Thank you!