

Week 03

IT-230

- Changing the Shell Environment
- Writing Bash Scripts
- Loops and Conditionals



IT-230 • Linux Administration

Start `workstation` and `servera` on your VDI



IT-230 • Linux Administration

Shell Variables

Values with a Name

Variables Explained

A variable is a **name** holding a **value**

Assign with **name=value** , then read it back with **\$name**

```
1 student@servera:~$ course=IT-230
2 student@servera:~$ echo $course
3 IT-230
4 student@servera:~$ course = IT-230
5 bash: course: command not found
6 student@servera:~$
```

No spaces are allowed around the **=**

Warning

With spaces Bash reads **course** as a command name, and **=** and **IT-230** as its arguments.

The Environment

`env` lists the variables that are available to **every executable you run**

A terminal window titled 'student@servera:~' with standard window controls. The terminal shows the command 'env' being executed, followed by a list of environment variables: HOME, LANG, LOGNAME, PATH, PWD, SHELL, and USER, followed by an ellipsis.

```
1 student@servera:~$ env
2 HOME=/home/student
3 LANG=en_US.UTF-8
4 LOGNAME=student
5 PATH=/home/student/.local/bin:/home/student/bin:/usr/local/bin:/usr/bin
6 PWD=/home/student
7 SHELL=/bin/bash
8 USER=student
9 ...
```

These are called **Environment Variables** because they are **exported** to subshells

Exported Variables

To make your variable available in subshells, you must **export** it

```
1  #^ 1. A variable you assign is local
2  student@servera:~$ course=IT-230
3  student@servera:~$ echo "$course"
4  IT-230
5  #^ 2. A subshell does not have it
6  student@servera:~$ bash
7  student@servera:~$ echo "$course"
8
9  student@servera:~$ exit
10 #^ 3. export makes it available to subshells
11 student@servera:~$ export course
12 student@servera:~$ bash
13 student@servera:~$ echo "$course"
14 IT-230
```

Each Executable runs in a SubShell

Every executable is ran from a subshell

If you want a variable to be available when executables are ran, **they must be exported**

Over time some programs have begun to rely on certain variables being exported

Two common ones are **PAGER** and **EDITOR**

PAGER and EDITOR

PAGER

Which program is used to display long output

Used by:

- `man`
- `git log`
- `systemctl status`

```
1 export PAGER=cat
2 man ls
```

EDITOR

Which program is used to edit text

Used by:

- `crontab -e`
- `visudo`
- `sudoedit`

```
1 export EDITOR=nano
2 sudoedit /etc/hosts
```

Exercise: Changing the Pager and Editor

GOAL

Choose which programs open your manual pages and your text

ENVIRONMENT

Hosts: `workstation`, then `servera`

WORKFLOW

1. Confirm that neither `PAGER` nor `EDITOR` is set
2. Export `PAGER` so that `man` prints straight to the terminal
3. Install `nano` and export `EDITOR` so it opens instead of `vi`
4. Confirm both are gone in a new session

Exercise: Changing the Pager and Editor

```
#####  
Exercise: Changing the Pager and Editor  
Requirements  
  Hosts: workstation, then servera  
Tasks  
  1. Confirm neither PAGER nor EDITOR is set  
  2. Export PAGER so man prints straight to the terminal  
  3. Install nano and export EDITOR so it opens instead of vi  
  4. Confirm both are gone in a new session  
#####  
[student@servera ~]$ █
```



[Written exercise](#)



IT-230 • Linux Administration

The Shell Prompt

Configuring the `PS1` Variable

PS1 Variable

What is displayed in your Bash prompt is configurable by changing the **PS1** variable

```
1 student@servera:~$ PS1='> '
2 > echo "Hello World"
3 Hello World
4 > PS1='servera: '
5 servera:
```

Dynamic Content in **PS1** Variable

Bash provides special character sequences. The ones below build the prompt you have been using

`student@workstation:~$`

`PS1=' \u @ \h : \w \b$ '`

A # for root and a \$ for everyone else

Shortening the Working Directory

The default is `\w` which only shows the last segment in the current working directory

Change it to `\w` for the full path

```
1 student@servera:log$ pwd
2 /var/log
3 student@servera:log$ echo $PS1
4 \u@\h:\w\ $
5 student@servera:log$ PS1='\u@\h:\w\ $ '
6 student@servera:/var/log$
```

Sequence	Working directory
<code>\w</code>	Full path
<code>\W</code>	Basename only

Finding Every Sequence

There are many more. Search the **PROMPTING** section of the Bash manual

```
1 man bash
2 /PROMPTING
```

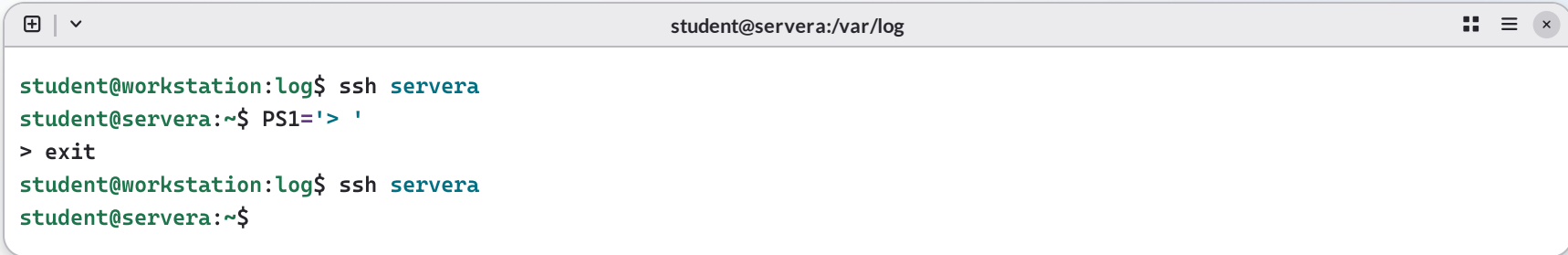
```
1 ...
2 \h    the hostname up to the first '.'
3 \H    the hostname
4 \j    the number of jobs currently managed by the shell
5 \t    the current time in 24-hour HH:MM:SS format
6 \T    the current time in 12-hour HH:MM:SS format
7 \@    the current time in 12-hour am/pm format
8 \u    the username of the current user
9 \v    the version of bash (e.g., 2.00)
10 \W    the basename of $PWD, with $HOME abbreviated with a tilde
11 \!    the history number of this command
12 ...
```

Settings Do Not Persist After Logout

Setting variables with `PS1=...` syntax does not survive a logout and back in

`PS1` and many other variables are set in configuration files

When you login, those files are read and variables are set to whatever value they contain



```
student@servera:/var/log
1 student@workstation:log$ ssh servera
2 student@servera:~$ PS1='> '
3 > exit
4 student@workstation:log$ ssh servera
5 student@servera:~$
```

To persist the changes you must add them to a configuration file

Persisting Changes

`~/.bashrc` is that file

Bash reads it every time you open a new terminal

Shell customizations you want to keep should be saved in `~/.bashrc`

```
1 ~/.bashrc
PS1='\u@\h:\w\$ '
```

Exercise: Customizing the Prompt

GOAL

Change your prompt for one shell, then keep the change

ENVIRONMENT

Hosts: `workstation` , then `servera`

WORKFLOW

1. Read the current value of `PS1`
2. Change `\W` to `\w` to show the full working directory
3. Move into a deeply nested directory and judge the result
4. Confirm the change disappears in a new session
5. Persist the prompt you prefer in `~/.bashrc`

Exercise: Customizing the Prompt

```
#####  
Exercise: Customizing the Prompt  
Requirements  
  Hosts: workstation, then servera  
Tasks  
  1. Read the current value of PS1  
  2. Change \W to \w to show the full working directory  
  3. Judge the result from a deeply nested directory  
  4. Confirm the change disappears in a new session  
  5. Persist the prompt you prefer in ~/.bashrc  
#####  
[student@servera ~]$ █
```



[Written exercise](#)



IT-230 • Linux Administration

The Shell Environment

A closer look at the `bashrc`

A Closer Look at `~/.bashrc`

The `~/.bashrc` file is loaded every time a new shell is launched

The `bashrc` contains `shell` code

A `bashrc` file will typically contain:

- Source declarations (imports)
- Variables
- Aliases
- Custom Functions

Sourcing other Files

Typically the first thing a `bashrc` does is import the settings from `/etc/bashrc`

```
~/ .bashrc
1 # Source global definitions
2 if [ -f /etc/bashrc ]; then
3     . /etc/bashrc
4 fi
```

The `.` is a synonym for the `source` command

The command `. /etc/bashrc` means:

- Execute all the code from `/etc/bashrc` as if it was typed here

Propagating System Settings

Since every user's `~/.bashrc` starts by sourcing `/etc/bashrc`

Customizations in that file end up being applied to all users

File	Applies to
<code>/etc/bashrc</code>	every user
<code>~/.bashrc</code>	you alone

bashrc Also Sets Your PATH

```
1 student@servera:~$ echo $PATH
2 /home/student/.local/bin:/home/student/bin:/usr/local/bin:/usr/bin:/usr/local/sbin:/usr/sbin
3 student@servera:~$
```

PATH is a list of directories separated by a colon **:**

When you type **ls** Bash searches the **PATH** directories from **left to right**

The first one holding an executable named **ls** wins

The **PATH** Variable

~/ .bashrc

```
1  ...
2  # User specific environment
3  if ! [[ "$PATH" =~ "$HOME/.local/bin:$HOME/bin:" ]]; then
4      PATH="$HOME/.local/bin:$HOME/bin:$PATH"
5  fi
6  export PATH
7  ...
```

The highlighted line is why **/home/student/bin** is on your **PATH**

export PATH is there so every program you run inherits it

Note

Don't worry about the **if** yet, we will learn that tonight!

Adding a Directory for This Shell

The change is not persisted when you open a new session

```
1  #^ 1. Append a directory, keeping what was already there
2  student@servera:~$ mkdir -p ~/scripts
3  student@servera:~$ PATH=$PATH:~/scripts
4  student@servera:~$ echo $PATH
5  /home/student/.local/bin:...:/usr/sbin:/home/student/scripts
6  #^ 2. Log out and back in and the change is gone
7  student@servera:~$ exit
8  student@workstation:~$ ssh servera
9  student@servera:~$ echo $PATH
10 /home/student/.local/bin:/home/student/bin:/usr/local/bin:/usr/bin:/usr/local/sbin:/usr/sbin
```

Splitting Settings Into Multiple Files

The default `~/.bashrc` contains this block of code

It will automatically source any file from the `~/.bashrc.d` directory

```
~/.bashrc
1  if [ -d ~/.bashrc.d ]; then
2      for rc in ~/.bashrc.d/*; do
3          if [ -f "$rc" ]; then
4              . "$rc"
5          fi
6      done
7  fi
8  unset rc
```

Warning

You cannot read this yet. By the end of today you will: it is a loop, two tests, and the `.` you just met.

Exercise: Putting Scripts on the `PATH`

GOAL

Give yourself a scripts directory that every new session can find

ENVIRONMENT

Hosts: `workstation` , then `servera`

WORKFLOW

1. Inspect the current `PATH` and find where it is already set in `~/.bashrc`
2. Create a `scripts` directory and append it to `PATH` for this shell only
3. Confirm it disappears in a new session
4. Add the same line to `~/.bashrc`
5. Confirm a new session now finds the directory

Exercise: Putting Scripts on the PATH

```
#####  
Exercise: Putting Scripts on the PATH  
Requirements  
  Hosts: workstation, then servera  
Tasks  
  1. Inspect the current PATH  
  2. Find where PATH is already set in ~/.bashrc  
  3. Create ~/scripts and add it to PATH for this shell only  
  4. Confirm it disappears in a new shell  
  5. Persist it in ~/.bashrc  
#####  
student@servera:~$ █
```



[Written exercise](#)



IT-230 • Linux Administration

Aliases

Shortcuts for long and frequently used commands

Aliases

Aliases allow you to define shortcuts for long or frequently used commands

Common Examples

```
1 alias ls='ls --color=auto'
2 alias sl='ls'
3 alias ..='cd ..'
4 alias tf='podman run --rm -it -v $(pwd):/workspace:Z -w /workspace hashicorp/terraform:latest'
```

Aliases are created using the `alias` builtin

```
alias lt='ls -lAtr'
```

**The command it expands into.
Must be quoted so the shell stores it intact**

Where Aliases Apply

- Expanded only when the name is the **first word** of a command
- Defined in the shell itself
- Lives in the **current shell only** until you make it permanent

Aliases you might define

```
1 alias sl='ls'  
2 alias ..='cd ..'  
3 alias rm='rm -i'
```



Warning

No spaces are allowed on either side of the equal sign.

Listing Aliases

Run `alias` with no arguments to see every alias in the current shell

```
1 student@servera:~$ alias
2 alias grep='grep --color=auto'
3 alias l.='ls -d .* --color=auto'
4 alias ll='ls -l --color=auto'
5 alias ls='ls --color=auto'
6 ...
7 student@servera:~$
```

RHEL predefines only a handful, mostly `--color` wrappers

Bypassing and Removing an Alias

The **unalias** command removes the shortcut entirely

```
student@servera:/tmp/at

#^ 1. An alias may shadow a real command of the same name
student@servera:/tmp/at$ alias grep='grep -i'
student@servera:/tmp/at$ grep alpha notes.txt
Alpha
alpha

#^ 2. Bypass it once with a backslash
student@servera:/tmp/at$ \grep alpha notes.txt
alpha

#^ 3. Remove it entirely with unalias
student@servera:/tmp/at$ unalias grep
student@servera:/tmp/at$ grep alpha notes.txt
alpha
```

Aliases Wrap Complex Commands

Run **terraform** from a container without installing it locally

```
1 podman run --rm -it -v $(pwd):/workspace:Z -w /workspace hashicorp/terraform:latest
```

Easy, right? Now type that ten to fifty times a day, every day

Or name it once

```
1 alias tf='podman run --rm -it -v $(pwd):/workspace:Z -w /workspace hashicorp/terraform:latest'
```

Persisting Aliases

Aliases could go in `~/.bashrc` , but they are easier to find in a file of their own

```
1 student@servera:~$ mkdir -p ~/.bashrc.d
2 student@servera:~$ echo "alias sl='ls'" >> ~/.bashrc.d/aliases
3 student@servera:~$ exit
4 student@workstation:~$ ssh servera
5 student@servera:~$ sl
6 scripts
```

No editing of `~/.bashrc` at all: that loop at the bottom loads the file for you

Exercise: Aliasing Away Typos

GOAL

Turn a typo you keep making into a working command, in every shell from now on

ENVIRONMENT

Hosts: `workstation`, then `servera`

WORKFLOW

1. List the aliases the shell already defines
2. Define an alias for a typo you make often and confirm it expands
3. Bypass the alias once without removing it
4. Confirm the alias is gone in a new session
5. Put it in `~/.bashrc.d/aliases` and confirm a new session loads it

Exercise: Aliasing Away Typos

Exercise: Aliasing Away Typos

Requirements

Hosts: workstation, then servera

Tasks

1. List the aliases the shell already defines
2. Alias a typo you make often
3. Bypass the alias once
4. Confirm the alias is gone in a new shell
5. Put it in ~/.bashrc.d/aliases

```
student@servera:~$
```



00:00



[Written exercise](#)



IT-230 • Linux Administration

Shell Options

Switches that change how Bash behaves

Toggling Bash Options with `set`

```
1 student@servera:~$ set -o
2 allexport      off
3 braceexpand   on
4 emacs         on
5 errexit       off
6 histexpand    on
7 history       on
8 interactive-comments  on
9 monitor       on
10 noclobber     off
11 ...
```

`set -o <option>` turns one on and `set +o <option>` turns it back off

The `noclobber` Option

Appending with `>>` still works, and `>|` still overwrites on purpose

```
1  #^ 1. noclobber refuses to overwrite an existing file
2  student@servera:~$ echo "Important Data" > file.txt
3  student@servera:~$ set -o noclobber
4  student@servera:~$ echo "New Data" > file.txt
5  -bash: file.txt: cannot overwrite existing file
6  #^ 2. Appending and forcing still work
7  student@servera:~$ echo "New Data" >> file.txt
8  student@servera:~$ echo "Force it" >| file.txt
```

Persisting Shell Options

To make the setting persistent, save it in `~/.bashrc`

```
1 ~/.bashrc  
  set -o noclobber
```

Exercise: Preventing Clobber

GOAL

Protect an existing file from being overwritten, in this shell and in every shell

ENVIRONMENT

Hosts: `workstation` , then `servera`

WORKFLOW

1. Create a file holding data you care about
2. Enable `noclobber` and attempt to overwrite the file with `>`
3. Add to the file with `>>` , then overwrite it deliberately with `>|`
4. Confirm the option is off again in a new session
5. Persist `noclobber` in `~/.bashrc` and confirm it applies at next login

Exercise: Preventing Clobber

```
#####  
Exercise: Preventing Clobber  
Requirements  
  Hosts: workstation, then servera  
Tasks  
  1. Create a file holding data you care about  
  2. Enable noclobber and try to overwrite it  
  3. Append with >> and overwrite deliberately with >|  
  4. Confirm the option is off again in a new shell  
  5. Persist noclobber in ~/.bashrc  
#####  
student@servera:~$ █
```



[Written exercise](#)



IT-230 • Linux Administration

Bash Scripts

Commands in a sequence

Refresher: `.` and `..`

Where does each file end up?

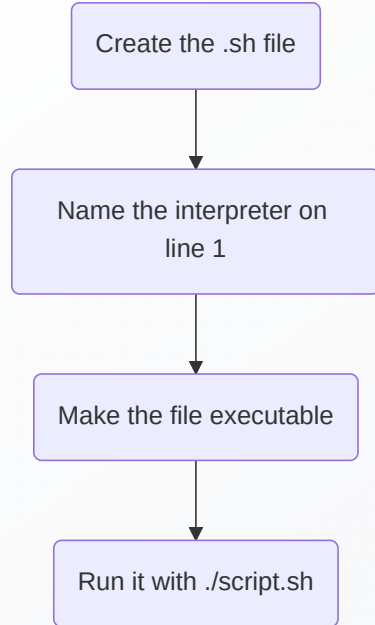
```
1 student@servera:~$ cd ~/bin
2 student@servera:~/bin$ mv ~/Downloads/engager.sh .
```

```
1 student@servera:~/bin$ pwd
2 /home/student/bin
3 student@servera:~/bin$ mv engager.sh ..
```

`.` is the directory I am in → `~/bin`

`..` is the directory above it → `/home/student`

A Script Is Just Commands in a File



take_inventory.sh

```
1 #!/bin/bash
2 echo "Taking inventory"
3 ls > inventory
```

Line 1 is the **hashbang** `#!`

It tells the kernel which interpreter should read the rest of the file

The remaining lines are commands that will be ran one after the other

Running a Shell Script

```
1 student@servera:~/scripts$ vim take_inventory.sh
2 student@servera:~/scripts$ bash take_inventory.sh
3 Taking inventory
4 student@servera:~/scripts$ ./take_inventory.sh
5 -bash: ./take_inventory.sh: Permission denied
6 student@servera:~/scripts$ chmod a+x take_inventory.sh
7 student@servera:~/scripts$ ./take_inventory.sh
8 Taking inventory
9 student@servera:~/scripts$
```

`bash script.sh` runs it without the execute bit. `./script.sh` requires it

Why Doesn't It Run by Name?

`~/scripts` is on the list, because you put it there this morning

```
1  #^ 1. Executable, but not found
2  student@servera:~$ ls -l take_inventory.sh
3  -rwxr-xr-x. 1 student student 51 Aug 28 10:52 take_inventory.sh
4  student@servera:~$ take_inventory.sh
5  -bash: take_inventory.sh: command not found
6  #^ 2. Where does Bash actually look?
7  student@servera:~$ echo $PATH
8  /home/student/.local/bin:...:/usr/sbin:/home/student/scripts
9  #^ 3. Move it somewhere on the PATH
10 student@servera:~$ mv take_inventory.sh ~/scripts
11 student@servera:~$ take_inventory.sh
12 Taking inventory
```

./ says **run the file at this path**
a bare name says **go find it on PATH**

Exercise: Our First Bash Script

GOAL

Write a script, make it runnable, and get it to run by name from anywhere

ENVIRONMENT

Host: `servera`

WORKFLOW

1. Write the script in your home directory with a hashbang
2. Try to run it, then add the execute permission
3. Run it with `./`
4. Find out why the bare name does not work
5. Move it onto your `PATH` and run it by name

Exercise: Our First Bash Script

```
#####  
Exercise: Our First Bash Script  
Requirements  
Host: servera  
Script location: /home/student/scripts/take_inventory.sh  
The script records what is in a directory  
Tasks  
1. Write the script in your home directory  
2. Try to run it and fix the permission  
3. Run it with ./  
4. Find out why the bare name does not work  
5. Move it onto your PATH and run it by name  
#####  
student@servera:~$
```



[Written exercise](#)



IT-230 • Linux Administration

Getting Data Into a Script

Variables, input, and arguments

Variables

A name holding a value you can use later

What is printed?

```
1 switch_ip='172.25.250.1'
2 echo "$switch_ip"
```

Output

```
1 172.25.250.1
```

And here?

```
1 switch_ip='172.25.250.1'
2 switch_ip='172.25.250.2'
3 echo "$switch_ip"
```

Output

```
1 172.25.250.2
```

An assignment replaces the old value, so the last one wins

The **read** Command

read pauses the script and waits for a line from **stdin**, then stores it in a variable

greeter.sh

```
1 #!/bin/bash
2 echo "Enter your last name:"
3 read -r last_name
4 echo "Hello $1 $last_name from $0"
```

[student@server ~]\$

- **-r** treats backslashes literally, so use it every time
- The script stops until a full line arrives
- **stdin** is normally the keyboard

Standard Output and Standard Input

> redirects stdout

Standard output normally goes to the terminal

> changes the destination to a file

```
1 echo "Student Extraordinaire" > file_name.txt
```

Instead of printing to the screen, print to a file

< redirects stdin

Standard input normally comes from the keyboard

< changes the source to a file

```
1 ./greeter.sh < file_name.txt
```

Instead of me typing it, read it from here

Feeding **read** from a File

I

The script cannot tell the difference. It just reads a line from **stdin**

Arguments

Values handed to the script on the command line

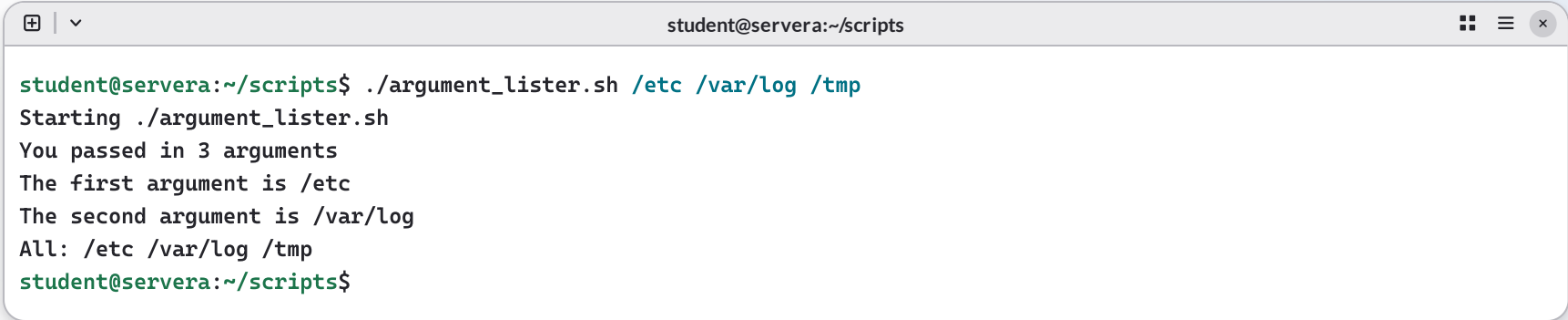
Variable	Holds
<code>\$0</code>	The script name
<code>\$1</code>	The first argument
<code>\$2</code>	The second argument
<code>\$3</code>	The third argument
<code>\$#</code>	How many arguments
<code>\$@</code>	All of them

argument_lister.sh

```
1 #!/bin/bash
2 echo "Starting $0"
3 echo "You passed in $# arguments"
4 echo "The first argument is $1"
5 echo "The second argument is $2"
6 echo "All: $@"
```

Arguments in Action

Bash substitutes each value before the script runs

A terminal window titled 'student@servera:~/scripts' with standard window controls. It shows the execution of a script with three arguments. The output shows the script recognizing the arguments and listing them.

```
1 student@servera:~/scripts$ ./argument_lister.sh /etc /var/log /tmp
2 Starting ./argument_lister.sh
3 You passed in 3 arguments
4 The first argument is /etc
5 The second argument is /var/log
6 All: /etc /var/log /tmp
7 student@servera:~/scripts$
```

Command Substitution

basename strips a path down to the file name

```
1 student@servera:~$ basename /etc/httpd/conf/httpd.conf
2 httpd.conf
3 student@servera:~$
```

Wrap a command in `$( )` and Bash runs it first, then substitutes its output

```
1 student@servera:~$ file_name=$(basename /etc/httpd/conf/httpd.conf)
2 student@servera:~$ echo "$file_name"
3 httpd.conf
4 student@servera:~$
```

Exercise: Passing Data to a Script

GOAL

Build a second script and feed it data as an argument, as typed input, and from a file

ENVIRONMENT

Host: `servera`

WORKFLOW

1. Create a new script from scratch and make it executable
2. Take the technician name as an argument
3. Read the starship name from `stdin`
4. Supply that input from a file instead of the keyboard
5. Add a value produced by another command

Exercise: Passing Data to a Script

```
#####  
Exercise: Passing Data to a Script  
Requirements  
Host: servera  
Script location: /home/student/scripts/shift_report.sh  
The script records who worked a shift and which system they touched  
Tasks  
1. Create a new script from scratch and make it executable  
2. Take the technician name as an argument  
3. Read the starship name from standard input  
4. Supply that input from a file instead of the keyboard  
5. Add a value produced by another command  
#####  
student@servera:~$
```



[Written exercise](#)



IT-230 • Linux Administration

For Loops

Administrators rarely manage one thing

Anatomy of a **for** Loop

Syntax

```
1 for VARIABLE in LIST; do
2     COMMAND VARIABLE
3 done
```

The variable takes each value in the list, one at a time

Visualization

```
1 for serverc in ; do
2     echo "serverc"
3 done
```

Output

```
1 servera
2 serverb
3 serverc
```

One Loop, Three Iterations

The loop is shorthand for writing the body out once per value

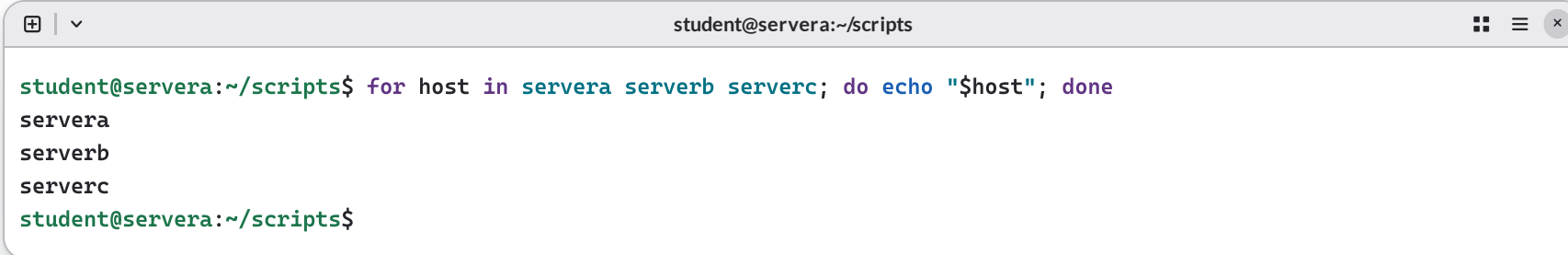
The loop

```
1 for host in servera serverb serverc; do
2     echo "$host"
3 done
```

What it amounts to

```
1 host=servera
2 echo "$host"
3 host=serverb
4 echo "$host"
5 host=serverc
6 echo "$host"
```

Watching It Iterate

A terminal window titled 'student@servera:~/scripts' with standard window controls. It shows a shell prompt and a for loop command. The loop iterates over the list 'servera serverb serverc', printing each item on a new line. The prompt returns after the last iteration.

```
student@servera:~/scripts$ for host in servera serverb serverc; do echo "$host"; done
servera
serverb
serverc
student@servera:~/scripts$
```

The body runs once per item, and the loop ends when the list runs out

Why Bother?

Ping the first fifteen hosts on `172.25.250.0/24`

By hand

```
1 ping -c1 -W1 172.25.250.1
2 ping -c1 -W1 172.25.250.2
3 ping -c1 -W1 172.25.250.3
4 ping -c1 -W1 172.25.250.4
5 ...
```

Twelve more lines, and one typo
away from a wrong answer

With a loop

```
1 for i in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15; do
2     ping -c1 -W1 172.25.250.$i
3 done
```

With brace expansion

```
1 for i in {1..15}; do
2     ping -c1 -W1 172.25.250.$i
3 done
```

Looping Over Arguments

"\$@" contains a list of all arguments passed into a script

A script can loop over this list directly

```
backup_file.sh
1 #!/bin/bash
2 for arg in "$@"; do
3     echo "Backing up $arg..."
4     cp "$arg" "$arg.bak"
5 done
```

Note

Always quote "\$@" so file names containing spaces survive as single items

```
student@servera:~$ ./backup_file.sh data.txt image.jpg
1 student@servera:~$ ./backup_file.sh data.txt image.jpg
2 Backing up data.txt...
3 Backing up image.jpg...
4 student@servera:~$
```

The same script handles one file or fifty

The caller decides

Refresher: Running a Command on Another Host

Interactive session

```
1 student@servera:~$ ssh student@serverb
2 student@serverb:~$
```

You land on the remote host and stay there

One command, then return

```
1 student@servera:~$ ssh student@serverb "hostname -f"
2 serverb.lab.example.com
3 student@servera:~$
```

Perfect inside a loop, because the output comes back to you

Exercise: Collecting Disk Data

GOAL

Gather disk information from every host named on the command line into one file per host

ENVIRONMENT

Hosts: `servera` and `workstation`

WORKFLOW

1. Create a basic script file and run it with `bash`
2. Make it executable and run it with `./`
3. Loop over a fixed list of hosts and confirm the loop runs once per host
4. Get the disk commands working on one host, then move them into the loop
5. Replace the fixed list with `"$@"` so the caller chooses the hosts
6. Verify the collected files and remove them

Exercise: Collecting Disk Data

#####

Exercise: Collecting Disk Data

Requirements

Hosts: servera and workstation
Script location: /home/student/scripts/collect_disk_data.sh
The script must run when called by name
The script must accept any number of host names as arguments
Run lsblk and df -h on each host
Store the output in /tmp/disk-data/{HOST}-disk-data.txt

Tasks

1. Create a basic script file
2. Make it executable
3. Write a basic for loop
4. Iterate on the logic
5. Make the script take arguments
6. Clean up



#####

student@servera:~\$ █



00:00



[Written exercise](#)



IT-230 • Linux Administration

Conditionals

Teaching a script to make a decision

Exit Codes

Every command reports whether it worked. The value is not printed, so ask for it with `$?`

The numbers

- Range `0` to `255`
- `0` means **success**
- Above `0` means **something went wrong**

```
1 student@servera:~$ ping -c1 -W1 172.25.250.10
2 ...
3 1 packets transmitted, 1 received, 0% packet loss
4 student@servera:~$ echo $?
5 0
```

A failure

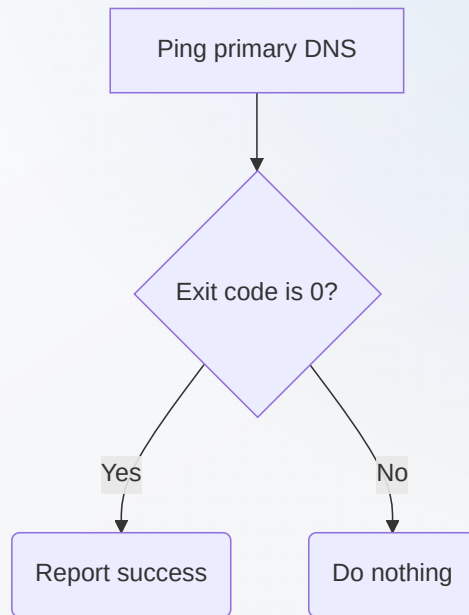
```
1 student@servera:~$ ping -c1 -W1 19.19.19.19
2 ...
3 1 packets transmitted, 0 received, 100% packet loss
4 student@servera:~$ echo $?
5 1
```

if ... then

The condition is a **command**, and its exit code is the answer

```
1 if <CONDITION>; then
2   <STATEMENT>
3 fi
```

```
1 primary_dns='8.8.8.8'
2 if ping -c1 -W1 "$primary_dns"; then
3   echo "Primary DNS reachable"
4 fi
```

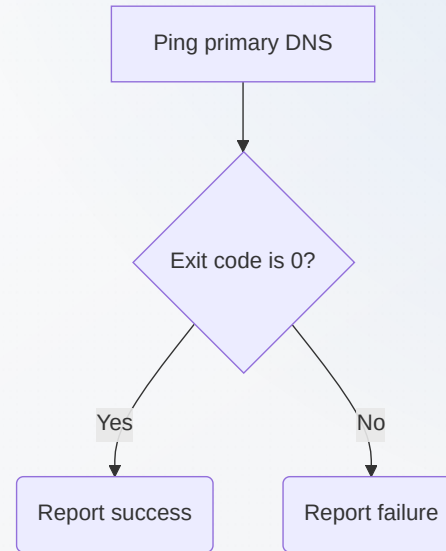


if ... then ... else

Now every outcome is handled

```
1 if <CONDITION>; then
2   <STATEMENT>
3 else
4   <STATEMENT>
5 fi
```

```
1 primary_dns='8.8.8.8'
2 if ping -c1 -W1 "$primary_dns"; then
3   echo "Primary DNS reachable"
4 else
5   echo "Primary DNS unreachable"
6 fi
```

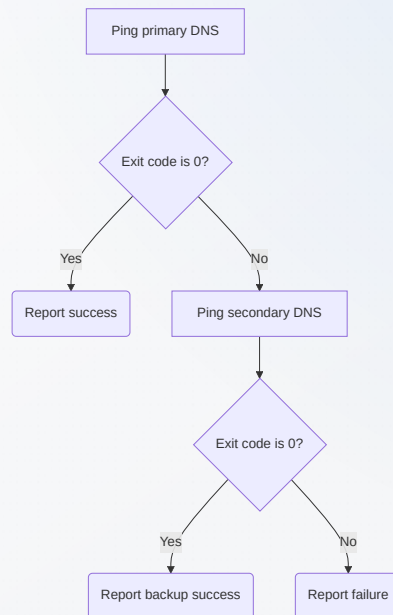


if ... elif ... else

elif is only reached when the test above it failed

```
1 if <CONDITION>; then
2     <STATEMENT>
3 elif <CONDITION>; then
4     <STATEMENT>
5 fi
```

```
1 if ping -c1 -W1 8.8.8.8; then
2     echo "Primary DNS reachable"
3 elif ping -c1 -W1 8.8.4.4; then
4     echo "Backup DNS reachable"
5 else
6     echo "Both unreachable"
7 fi
```



Exercise: Determining Host Status

GOAL

Report which hosts on a list are reachable and which are not

ENVIRONMENT

Host: `servera`

WORKFLOW

1. Create a basic script file
2. Make it executable and run it
3. Loop over a list of addresses and announce each one as it is checked
4. Branch on the result of `ping` to report each host as up or down
5. Silence the `ping` output so only your own report remains
6. Confirm that both a reachable and an unreachable address are reported correctly

Exercise: Determining Host Status

```
#####  
Exercise: Determining Host Status  
Requirements  
Host: servera  
Script location: /home/student/scripts/pinger.sh  
Report which addresses used by the finance application are reachable  
Tasks  
1. Create a basic script file  
2. Make it executable  
3. Write a basic for loop  
4. Branch on the result of ping  
5. Silence the ping output  
6. Clean up  
#####  
student@servera:~$
```



[Written exercise](#)

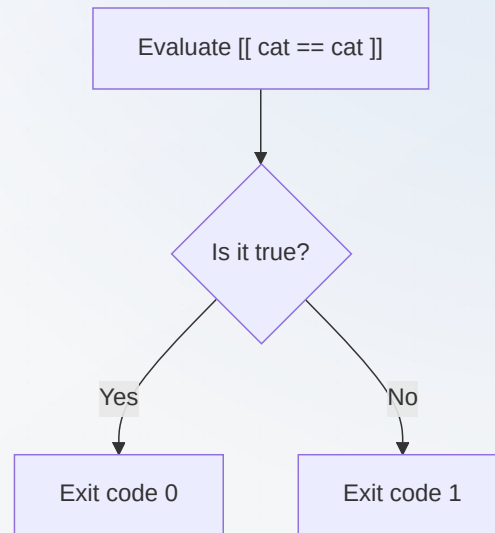
The Enhanced Test Command `[[ ]]`

An exit code is exactly what `if` was already reading, so the two fit together

`[[ ... ]]` evaluates an expression and reports the answer as an **exit code**

- Expression **true** → exit code `0`
- Expression **false** → exit code `1`

```
1 student@servera:~$ [[ "cat" == "cat" ]]
2 student@servera:~$ echo $?
3 0
4 student@servera:~$ [[ "cat" == "dog" ]]
5 student@servera:~$ echo $?
6 1
7 student@servera:~$
```



Comparing Strings

The full operator list is in `man bash` under `/CONDITIONAL EXPRESSIONS`

Equality

Operator	Means	Example
<code>==</code>	Equal	<code>[["\$a" == "\$b"]]</code>
<code>!=</code>	Not equal	<code>[["\$a" != "\$b"]]</code>

Emptiness

Operator	Means	Example
<code>-z</code>	Empty	<code>[[-z "\$name"]]</code>
<code>-n</code>	Not empty	<code>[[-n "\$name"]]</code>

Guarding Against Missing Input

A script that assumes it was given an argument fails in confusing ways

```
1 if [[ -z "$1" ]]; then
2     echo "Error: missing argument"
3     exit 1
4 fi
```

Stop early with a clear message and a nonzero exit code

Warning

Always quote variables in a string comparison. An unquoted empty variable becomes a syntax error.

Comparing Numbers and Inspecting Files

Numbers

Operator	Means
<code>-eq</code>	Equal to
<code>-ne</code>	Not equal to
<code>-gt</code>	Greater than
<code>-lt</code>	Less than
<code>-ge</code>	Greater or equal
<code>-le</code>	Less or equal

Files and directories

Operator	Means
<code>-e</code>	Exists
<code>-f</code>	Is a regular file
<code>-d</code>	Is a directory
<code>-r</code>	Is readable
<code>-w</code>	Is writable
<code>-x</code>	Is executable

Negation with !

! flips the exit code, so the branch runs when the test **fails**

```
1 if ! ping -c1 -W1 172.25.250.99; then
2     echo "Host is down!"
3 fi
```

```
1 if ! [[ -e /tmp/report.txt ]]; then
2     echo "Report has not been generated yet"
3 fi
```

Note

Read ! as "not". **if ! ping** is "if pinging did not succeed".

Exercise: Item Inspector

GOAL

Report whether each item in a list is a file, a directory, or neither

ENVIRONMENT

Host: `servera`

WORKFLOW

1. Create a basic script file
2. Make it executable and run it
3. Loop over a list of paths and print each one
4. Report the paths that are regular files
5. Add a branch for directories, then a final branch for everything else
6. Confirm that every item in the list produces exactly one message

Exercise: Item Inspector

```
#####  
Exercise: Item Inspector  
Requirements  
Host: servera  
Script location: /home/student/scripts/backup_auditor.sh  
Inspect each path a backup job was told to collect  
Report each path as a regular file, a directory, or neither  
Every path must produce exactly one message  
Tasks  
1. Create a basic script file  
2. Make it executable  
3. Write a basic for loop  
4. Report the regular files  
5. Add branches for directories and everything  
6. Clean up  
#####  
student@servera:~$ █
```



[Written exercise](#)



IT-230 • Linux Administration

While Loops

Repeating until something changes

while Repeats As Long As the Condition Succeeds

Syntax

```
1 while CONDITION; do
2     COMMAND
3 done
```

A **for** loop knows its list up front, but a **while** loop does not

Waiting for a host to come back

```
1 while ! ping -c1 -W1 172.25.250.10; do
2     echo "Waiting for host..."
3     sleep 1
4 done
5 echo "Host is up!"
```

Output

```
1 Waiting for host...
2 Host is up!
```

Loops That Never End

A condition that is always true produces a loop that runs forever

```
1 while true; do
2     echo "Scanning for new files"
3     sleep 10
4 done
```

Stop it with `Ct r l+C`, or `kill` it from another shell

⚠ Warning

An accidental one fills a disk or pegs a CPU.

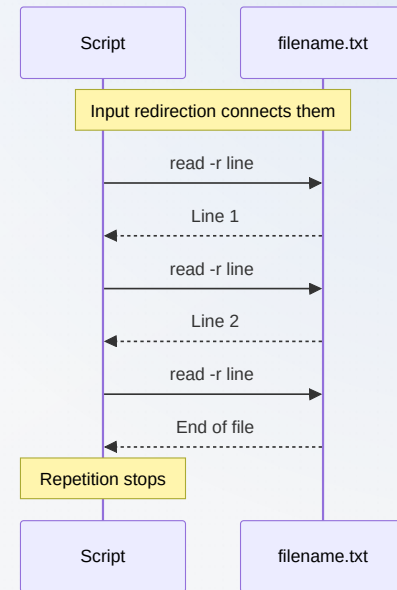


Reading a File Line by Line

The most common `while` loop an administrator writes

```
1 while read -r line; do
2     echo "Processing: $line"
3 done < filename.txt
```

- `< filename.txt` feeds the file into the loop's `stdin`
- `read -r line` takes one line and stores it in `$line`
- The body runs once per line



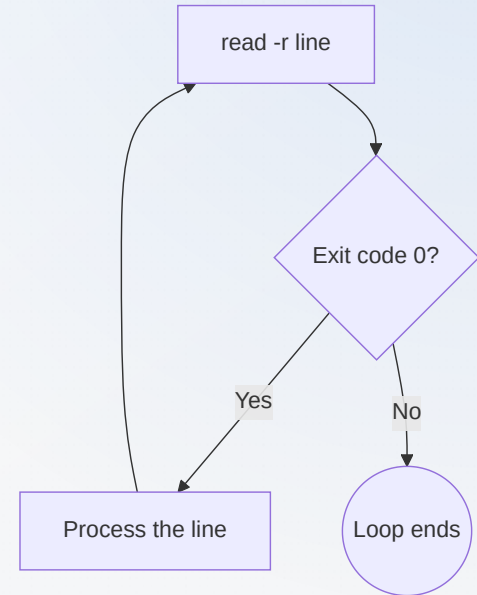
Why Does That Loop Ever Stop?

`read` is a command, so it reports an exit code like any other

- **Success, `0`**: it read a line
- **Failure, `1`**: it reached the end of the file

`while` keeps going as long as `read` returns `0`

The loop ends when the file has no more lines to process



Exercise: Creating Users from a File

GOAL

Create a user account for every name listed in an input file

ENVIRONMENT

Host: `servera`

WORKFLOW

1. Write the list of accounts to create into an input file
2. Create an executable script that reads the file line by line
3. Create each listed account and report whether it worked
4. Refuse to run unless the script was started with root privileges
5. Skip blank lines, then rerun the script to confirm it is safe to repeat
6. Remove the accounts the script created

Exercise: Creating Users from a File

```
#####  
Exercise: Creating Users from a File  
Requirements  
Host: servera  
Script location: /home/student/scripts/user_creator.sh  
Create an account for every name listed in new_users.txt  
The script must be safe to run twice  
Only a privileged user may run it  
Tasks  
1. Stage the new_users.txt file  
2. Create a basic script file and make it executable  
3. Read the file line by line  
4. Report whether each account was created  
5. Refuse to run without root privileges  
6. Skip blank lines  
7. Clean up  
#####  
student@servera:~$
```



00:00



[Written exercise](#)

Remember This?

Skip anything that is not a regular file, then source what is left

```
1 #^ The block at the bottom of ~/.bashrc
2 if [ -d ~/.bashrc.d ]; then
3     for rc in ~/.bashrc.d/*; do
4         if [ -f "$rc" ]; then
5             . "$rc"
6         fi
7     done
8 fi
9 unset rc
```

Note

`[ ]` is the older syntax for `[[ ]]`. RHA Material uses `[[ ]]`