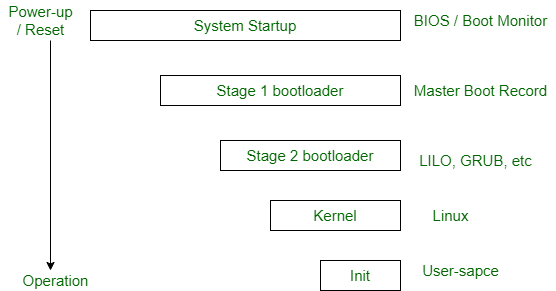
What is Booting

A computer without a program running is just an inert hunk of electronics. The first thing a computer has to do when it is turned on is to start up a special program called an operating system. The operating system’s job is to help other computer programs work by handling the messy details of controlling the computer’s hardware.

### An overview of the boot process



The boot process is something that happens every time you turn your computer on. You don’t really see it, because it happens so fast. You press the power button and come back a few sec (or minutes if on slow storage like HDD) later and Windows 10, or Windows 11, or whatever Operating System you use is all loaded.

The BIOS chip tells it to look in a fixed place, usually on the lowest-numbered hard disk (the boot disk) for a special program called a boot loader (under Linux the boot loader is called Grub or LILO). The boot loader is pulled into memory and started. The bootloader’s job is to start the real operating system.

A run level is a state of init and the whole system that defines what system services are operating. Run levels are identified by numbers. Some system administrators use run levels to define which subsystems are working, e.g., whether X is running, whether the network is operational, and so on.

* Whenever a LINUX system boots, firstly the **init** process is started which is actually responsible for running other start scripts which mainly involves initialization of you hardware, bringing up the network, starting the graphical interface.
* Now, the **init** first finds the default **runlevel** of the system so that it could run the start scripts corresponding to the default run level.
* A **runlevel** can simply be thought of as the state your system enters like if a system is in a single-user mode it will have a **runlevel 1** while if the system is in a multi-user mode it will have a **runlevel 5**.
* A **runlevel** in other words can be defined as a **preset single digit integer** for defining the operating state of your LINUX or UNIX-based operating system. Each runlevel designates a different system configuration and allows access to different combination of **processes**.

The important thing to note here is that there are differences in the runlevels according to the operating system. The standard **LINUX kernel** supports these seven different runlevels :

* 0 – System halt *i.e* the system can be safely powered off with no activity.
* 1 – Single user mode.
* 2 – Multiple user mode with no NFS(network file system).
* 3 – Multiple user mode under the command line interface and not under the graphical user interface.
* 4 – User-definable.
* 5 – Multiple user mode under GUI (graphical user interface) and this is the standard runlevel for most of the LINUX based systems.
* 6 – Reboot which is used to restart the system.

By default most of the LINUX based system boots to runlevel 3 or runlevel 5.  
In addition to the standard runlevels, users can modify the preset runlevels or even create new ones according to the requirement. Runlevels 2 and 4 are used for user defined runlevels and runlevel 0 and 6 are used for halting and rebooting the system.

Reboot:

**reboot** command is used restart or reboot the system. In a Linux system administration, there comes a need to restart the server after the completion of some network and other major updates. It can be of software or hardware that are being carried on the server. The reboot is needed so that the changes that the user have done can be affected on the server. For example, if the user is re-compiling the server’s kernel that is going through some more advanced server administration, then he needs to restart the machine in order to complete the compilation and to have a new updated kernel version on the server. When Updating the server’s memory, IP allocation, NIC configuration are the key tasks that need to be done on the server restarted once leading to their successful implementation. Most of the Linux system administrators access their servers via shell or SSH to perform a bunch of administrative activities, server management, and monitoring. So they need to know the basic commands to restart the server from the shell.

**Syntax:**

reboot [OPTIONS...]

**Options:**

* **–help :** This option prints a short help text and exit.
* **–halt :** This option halt the machine, regardless of which one of the three commands is invoked.
  + **-p, –poweroff :** This option will going to power-off the machine, regardless of which one of the three commands is being invoked.
  + **–reboot :** This option reboot the machine, regardless of which one of the three commands is invoked.
  + **-f, –force :**This option force immediate halt, power-off, or reboot. When it is specified once, this results in the immediate but clean shutdown by the system manager. When it is specified twice, this results in immediate shutdown without contacting the system manager. See the description of the option –force in systemctl(1) for more details.
* **-w, –wtmp-only :**This option only writes *wtmp*shutdown entry, it do not actually halt, power-off, reboot.

**Examples:**

**Restart your system:** If you just need is a restart without going into any details just help yourself by using any one of the following commands:

$sudo reboot

$sudo shutdown –r now

**Note**that the usage of the reboot, halt and poweroff is almost similar in syntax and effect. Run each of these commands with –help to see the details.

ShutDown:

# shutdown

The shutdown command brings down system in a secure way. All the logged-in users are notified about the system shutdown.

Signal SIGTERM notifies all the processes that the system is going down, so that processes can be saved and exit properly.

Command shutdown signals the init process to change the runlevel.

**Runlevel 0** halts the system

You can shutdown a system by passing a definite time (in minutes). System will automatically shutdown after specified minute giving a message and time to save all work.

**Syntax:**

1. shutdown **<time>**

**Example:**

shutdown 3

To immediately shutdown the system, use **now** option,

**Syntax:**

1. shutdown now

System will shutdown immediately.

**halt** - reboot or stop the system

## SYNOPSIS

halt [OPTION]...

## DESCRIPTION

These programs allow a system administrator to **reboot, halt or poweroff** the system.

When called with --force or when in runlevel 0 or 6, this tool invokes the reboot system call itself and directly reboots the system. Otherwise this simply invokes the shutdown tool with the appropriate arguments.

## Options

|  |  |
| --- | --- |
| **Tag** | **Description** |
| **-f, --force** | Does not invoke shutdown and instead performs the actual action you would expect from the name. |
| **-p, --poweroff** | Instructs the halt command to instead behave as poweroff. |
| **-w, --wtmp-only** | Does not call shutdown or the reboot system call and instead only writes the shutdown record to /var/log/wtmp |
| **--verbose** | Outputs slightly more verbose messages when rebooting, useful for debugging problems with shutdown. |

## EXAMPLES

**Example-1:**

The halt command will cease all CPU function on the system. On most systems, this will drop you into single-user mode and then power off the machine:

$ halt

output:  
Broadcast message from ubuntu@ubuntu  
root@ubuntu:/var/log#   (/dev/pts/0) at 7:38 ...  
  
The system is going down for halt NOW!

**Shell basics: Comparision of Shells, Variables, Shell Scripts, Arithmetic in Shell Script ,Looping**

Use below link

<https://www.slideshare.net/moayadmoawiah/shell-programming-10848411>

Linux Shell / Shell Script Best Practices

This page lists best practices, based on industry standards and first-hand experience.

* [Use simple shell scripts to memorialize tasks](https://learn.openwaterfoundation.org/owf-learn-linux-shell/best-practices/best-practices/#use-simple-shell-scripts-to-memorialize-tasks)
* [Use version control](https://learn.openwaterfoundation.org/owf-learn-linux-shell/best-practices/best-practices/#use-version-control)
* [Indicate the shell to run in the first line of shell scripts](https://learn.openwaterfoundation.org/owf-learn-linux-shell/best-practices/best-practices/#indicate-the-shell-to-run-in-the-first-line-of-shell-scripts)
* [Write code that is understandable](https://learn.openwaterfoundation.org/owf-learn-linux-shell/best-practices/best-practices/#write-code-that-is-understandable)
* [Check whether the script is running in the correct folder](https://learn.openwaterfoundation.org/owf-learn-linux-shell/best-practices/best-practices/#check-whether-the-script-is-running-in-the-correct-folder)
* [Echo useful troubleshooting information](https://learn.openwaterfoundation.org/owf-learn-linux-shell/best-practices/best-practices/#echo-useful-troubleshooting-information)
* [Consider options for logging](https://learn.openwaterfoundation.org/owf-learn-linux-shell/best-practices/best-practices/#consider-options-for-logging)
* [Create documentation](https://learn.openwaterfoundation.org/owf-learn-linux-shell/best-practices/best-practices/#create-documentation)
* [Include useful web resource links in shell script comments](https://learn.openwaterfoundation.org/owf-learn-linux-shell/best-practices/best-practices/#include-useful-web-resource-links-in-shell-script-comments)
* [Learn how to use shell features](https://learn.openwaterfoundation.org/owf-learn-linux-shell/best-practices/best-practices/#learn-how-to-use-shell-features)
* [Use functions to create reusable blocks of code](https://learn.openwaterfoundation.org/owf-learn-linux-shell/best-practices/best-practices/#use-functions-to-create-reusable-blocks-of-code)

Use simple shell scripts to memorialize tasks

It is often necessary to perform a number of steps to process data and/or automate program calls to complete a task. These steps may be a one-time task, but often will need to be repeated in the future. Rather than relying on written or electronic notes, email, etc., creating a short shell script to memorialize the task can ensure that knowledge is retained. The script can also be enhanced over time to provide more functionality.

A useful best practice is to create a simple shell script in the files being processed, with comments in the shell script to explain its purpose and use. Additionally, a README.md file can be created to provide formatted explanation of the shell script.

The script and README.md file can be managed in a version control system to track changes over time and serve as a backup.

Use version control

Any script worth creating and using is probably worth tracking in a version control system. There is nothing more frustrating that asking "where did I put that script?" Therefore, use Git and a cloud-hosted version control system like GitHub or Bitbucket to maintain the script. This also provides information about the author so that questions and bugs can be dealt with, for example via the repository issues page.

If a true version control system is not used, the script can also be saved in a knowledge base or other information platform.

Indicate the shell to run in the first line of shell scripts

A shell script (actually any script) can be written in various languages and language standards. Shell scripts can be written for sh, bash, or other command shells. Do not assume that the default command shell that will be used will be correct. Therefore, always specify the shell to run, for example:

#!/bin/sh

#

# The above indicates that the Bourne shell `sh` command shell will be used to run the script.

Write code that is understandable

It is common that code is rewritten simply because the original author's work is not understandable. This results in extra cost, potentially bugs, and potentially loss in functionality if the original code was not understood. The urge to rewrite code may be because of a lack of documentation, confusing logic, bad programming style, use of obscure or advanced language syntax without explanation, or other reasons.

Documenting code at the time it is being written is the best time to document code. If updating code, read the code comments again and if they do not make sense, clarify the comments. A simple rule is to ask "will the next person working on this code understand it?"

The following are some basic guidelines to making code understandable:

* Add full grammar comments to code. Don't force people to assume what you mean. Use proper grammar. Sloppy comments and incomplete thoughts can indicate sloppy code.
* Explain complex syntax. Don't assume that the next programmer will have a PhD in shell scripting. Yes, every answer can be found on the web, but the web also contains many misleading and out of date examples.
* Use variable and function names that are verbose enough to provide context. Code should read like a clear process. Using appropriate names will also reduce the need for code comments.
* Be consistent in names and style. If editing someone else's code, try to be consistent with the original style if possible.
* Use double or single quotes around strings, as appropriate, to indicate strings.
* Use appropriate indentation consistently. Tabs are OK and if used should not be mixed with spaces. Spaces if used should be in groups of 2 or 4. Do not assume that the next person to edit the code will use the same convention in their editor, and make it obvious what is being used by being consistent.

Check whether the script is running in the correct folder

The folder from which a script is run often has a large impact on the functionality of the script. Options include:

1. Allow the script to run in any folder since the task does not depend on the location.
2. Allow the script to run in any folder and locate input and output based on the environment, such as detecting the user's home folder or a standard folder structure within the environment.
3. Require the script to be run from a certain folder because input and output folders are relative to the run folder.

Great pains may need to be taken in a script's code to ensure that a script runs correctly in any folder, in order to ensure that input files are found and output is created in the proper location. However, for simple tasks, it may be easier to require running from a specific folder.

A best practice is to put checks in place, if necessary, to ensure that the script is running in the correct folder. For example, the following does a simple check to make sure that the script is running in a build-util folder. The following check is not fool-proof because running in one build-util folder and specifying a path to another build-util folder would pass the test, but the basic check helps prevent many issues.

# Make sure that this is being run from the build-util folder

pwd=`pwd`

dirname=`basename ${pwd}`

if [ ! ${dirname} = "build-util" ]

then

echo "Must run from build-util folder"

exit 1

fi

A more robust solution is to allow the script to be run from any folder, including finding in the PATH or specifying the path to the script manually. In this case the following syntax can be used to determine the location of the script, and other folders and files can be located relative to that location, assuming there is a standard.

# Get the location where this script is located since it may have been run from any folder

scriptFolder=`cd $(dirname "$0") && pwd`

# Also determine the script name, for example for usage and version.

# - this is useful to prevent issues if the script name has been changed by renaming the file

scriptName=$(basename $scriptFolder)

Echo useful troubleshooting information

Shell scripts can be difficult to troubleshoot, especially if the script coding is not clear and the script user did not write the script. Therefore, it is often helpful to print important configuration information at the start of the script. For example use echo statements to print important environment variable names and values, names of input files, etc.

If the script is more advanced, such output could be printed only when a command line parameter is specified, such as --verbose.

Consider options for logging

Logging for shell scripts can be implemented in various ways. The standard for Linux is often to output to the stdout stream (for example echo ...). The script output can then be redirected into a file or piped to another program for further processing. However, diagnostic or progress messages that are separate from analytical output generally need to be separated from the general output.

Depending on the complexity of the script, it may be very useful to save logging messages to a file. This can be done by echoing output to a file in the current folder or a special location. The author of the script probably has a good idea of how logging should be done because they use it themselves. A best practice is to implement options for logging in a way that will benefit users of the script and then provide documentation to explain options. This will help users, especially those who may not fully understand how to do logging with redirection.

The following is a basic example of implementing logging:

#!/bin/sh

# example-logging

# This example shows basic logging approach

# Define the logfile

# - in this case it is a temporary file but a name and location

# specific to the script purpose would normally be used

logFile="$(mktemp).log"

scriptName="myscript"

# Write one record to the logfile indicating the time and program

# - use tee command to show output to terminal and logfile if appropriate

# - write standard output and error to the file

now=$(date --iso-8601=seconds)

echo "[${now}] Logfile for $scriptName: $logFile" 2>&1 | tee $logFile

# Write subsequent message by appending

echo "Another log message" 2>&1 | tee --append $logFile

The script output is:

[2019-04-07T01:50:12-06:00] Logfile for myscript: /tmp/tmp.S25lMrx7nN.log

Another log message

Create documentation

A shell script is only truly useful if someone other than the original author can use it. Therefore, all shell scripts should have documentation in one or more forms, depending on the significance of the script, including:

* Code comments, enough to understand the purpose, and good in-line comments.
* A printUsage() or similar function that prints basic usage, run via -h or --help, or by default to show user available command options.
* README.md file in the repository.
* User documentation, such as this documentation.

Include useful web resource links in shell script comments

Technologies can be complicated and shell programming is no different. It is often necessary to search the web for a solution or comparative example to perform a task. Once the solution is coded, it can be difficult to understand the approach. Therefore, include a comment in the code with the link to the web resource that explained the solution. Don't make the next programmer relearn the same material from terse code. Allow the code to be a teaching tool that can be maintained by the next developer. Also, including links is a way to give credit to the person that helped solve a technical issue.

Learn how to use shell features

There is always a brute force "quick and dirty" way to do things, such as by copying and pasting code from a web search. However, there is a balance between the technical debt of quick and dirty solutions, and more elegant solutions that take more time to learn, but are more robust and maintainable in the long run. In particular, quick and dirty solutions that negatively impact the user experience and resources spent by other developers should be avoided. Shell programmers should take the time to learn shell programming concepts and features such as command line parsing, functions, log files, etc. so that they can improve shell script quality and functionality. This documentation is an attempt to provide easily understood examples that go beyond the often terse and trivial examples on the web.

Use functions to create reusable blocks of code

This should go without saying, but modular code tends to be easier to maintain, especially when a script becomes long. Every script author should consider using functions to organize script functionality. Functions also simplify sharing code between scripts since functions can be copied and pasted between scripts.

The alternative to writing a function is to write a separate script that can be called with an appropriate command line. This approach is OK as long as the called script is located in a location that can be found by the calling script.

**Introduction to Python:-**

Python is a high-level, general-purpose and a very popular programming language. Python programming language (latest Python 3) is being used in web development, Machine Learning applications, along with all cutting edge technology in Software Industry. Python Programming Language is very well suited for Beginners, also for experienced programmers with other programming languages like C++ and Java.

This specially designed Python tutorial will help you learn Python Programming Language in most efficient way, with the topics from basics to advanced (like Web-scraping, Django, Deep-Learning, etc.) with examples.

Below are some facts about Python Programming Language:

1. Python is currently the most widely used multi-purpose, high-level programming language.
2. Python allows programming in Object-Oriented and Procedural paradigms.
3. Python programs generally are smaller than other programming languages like Java. Programmers have to type relatively less and indentation requirement of the language, makes them readable all the time.
4. Python language is being used by almost all tech-giant companies like – Google, Amazon, Facebook, Instagram, Dropbox, Uber… etc.
5. The biggest strength of Python is huge collection of standard library which can be used for the following:
   * [Machine Learning](https://www.geeksforgeeks.org/machine-learning/)
   * GUI Applications (like [Kivy](https://www.geeksforgeeks.org/kivy-tutorial/" \t "_blank), Tkinter, PyQt etc. )
   * Web frameworks like [Django](https://www.geeksforgeeks.org/django-tutorial/) (used by YouTube, Instagram, Dropbox)
   * Image processing (like [OpenCV](https://www.geeksforgeeks.org/opencv-python-tutorial/), Pillow)
   * Web scraping (like Scrapy, BeautifulSoup, Selenium)
   * Test frameworks
   * Multimedia
   * Scientific computing
   * Text processing and many more..

**Similarities between Python and Shell scripting**

1. Interpreted Languages:

Both Python and shell scripts are interpreted languages, which means that the code is executed line by line without the need for compilation.

Python Example: # No need for compiling, just execute the code

print("Hello, World!")

Shell Script Example: # No need for compiling, just execute the script

#!/bin/bash echo "Hello, World!"

In both examples, the code can be executed without compiling. The Python code simply prints "Hello, World!" to the console, while the shell script does the same thing using the echo command.

2. Scripting:

Both Python and shell scripting are mainly used for scripting purposes, such as automation, data processing, and system administration.

Python Example:

# Scripting example in Python import os

# List all files in the current directory

dir\_list = os.listdir(".")

for filename in dir\_list: print(filename)

Shell Script Example:

# Scripting example in Shell

#!/bin/bash # List all files in the current directory

for filename in \*;

do echo $filename done

In both examples, the code is used to perform a task that involves multiple steps. In the Python example, the code uses the os module to get a list of files in the current directory and prints them to the console. In the shell script example, the code uses a loop to iterate over all files in the current directory and prints their names to the console.

3. Dynamically Typed:

both Python and shell scripting are dynamically typed languages, which means that the data type of a variable is determined at runtime and can be changed during the execution of the program. This is in contrast to strongly typed languages, which enforce strict type checking at compile time.

In Python and shell scripting, you can declare a variable and assign it a value without specifying its data type. The interpreter will automatically determine the data type based on the value assigned to the variable.

For example, in Python, you can create a variable x and assign it an integer value, then change its value to a string later:

Python Example

x = 10 print(x)

# output: 10

x = "Hello, World!" print(x)

# output: Hello, World!

In this example, x is initially assigned an integer value of 10, so the interpreter determines that it is an integer variable. Later, the value of x is changed to a string, so the interpreter determines that it is a string variable. Similarly, in shell scripting, you can create a variable y and assign it a string value, then change its value to an integer later:

Bash

y="Hello, World!"

echo $y

# output: Hello, World!

y=10

echo $y

# output: 10

In this example, y is initially assigned a string value, so the interpreter determines that it is a string variable. Later, the value of y is changed to an integer, so the interpreter determines that it is an integer variable.

4. File Manipulation:

Both languages support file manipulation, including reading from and writing to files.

In Python, you can use the built-in open function to open a file and perform various file operations, such as reading, writing, and appending data to a file.

Python

# Open the file for reading with

open('example.txt', 'r') as f:

# Read the contents of the file contents = f.read()

# Print the contents to the console print(contents)

In this example, we're using the open function to open a file called "example.txt" for reading. The file is opened in a context manager using the with statement, which ensures that the file is properly closed after we're done with it. We then read the contents of the file using the read method of the file object, and print the contents to the console.

In shell scripting, you can use various command-line utilities to perform file operations, such as cat, grep, sed, and awk.

bash

# Read the file line by line and print each line to the console

while read line

do

echo $line

done < filename.txt

In this example, we're using a while loop to read a file called filename.txt line by line. The read command reads a single line of input from standard input (in this case, the contents of the file), and assigns it to the line variable. The loop continues until all lines have been read.

Inside the loop, we're using the echo command to print each line to the console. The $line variable contains the current line of input, which is passed as an argument to the echo command.

Both of these examples demonstrate how you can manipulate files in Python and shell scripting. While Python has built-in file manipulation functions, shell scripting relies on external command-line utilities to perform file operations

5. Both can execute system commands:

We can execute system commands within the code itself.

In Python, you can use the subprocess module to execute system commands.

python import subprocess

# Run the "ls" command and print its output

result = subprocess.run(['ls'], stdout=subprocess.PIPE)

print(result.stdout.decode('utf-8'))

In shell scripting, you can simply use the name of the system command as a statement, and the shell will execute it.

#!/bin/bash

# Get the current date and time

now=$(date +"%Y-%m-%d %H:%M:%S")

# Create a new directory with the current date and time as the name

mkdir $now

# Copy a file to the new directory

cp myfile.txt $now

Here's a table comparing some of the differences between the Bourne shell (sh) and the C shell (csh):

| **Feature** | **Bourne shell (sh)** | **C shell (csh)** |
| --- | --- | --- |
| Syntax | Simple and straightforward | More complex and C-like syntax |
| Command history | Limited or absent | Available with command history substitutions |
| Environment variables | Accessed using **$** syntax | Accessed using **$** syntax or **setenv** command |
| Aliases | Available with **alias** command | Available with **alias** command |
| Command substitution | Available with **$(command)** syntax | Available with **`command`** or **$(command)** syntax |
| Shell scripts | Supported | Supported |
| Job control | Basic support with **&** and **ctrl-Z** | Advanced support with **fg**, **bg**, and **jobs** commands |
| File name completion | Basic support with **Tab** key | Advanced support with **Tab** key and file name expansion |
| Command-line editing | Limited or absent | Available with Emacs or vi-like keybindings |
| Logical operators | Available with **&&** and ` |  |
| Looping constructs | Available with **for** and **while** | Available with **foreach** and **while** |
| Scripting language features | Limited or absent | Available with arrays and other advanced features |
| Standard input/output | Supported | Supported |
| Scripting speed | Faster | Slower |