What is Shell?

A brief introduction to Shell and its types.

We'll cover the following

> A shell is a macro processor or a command language interpreter that primarily translates the commands, written by the user in the terminal, into system actions that are executed, which can also automatically run in programs called shell scripts.

A shell is not an operating system. It is a way to interface with the operating system and run commands.

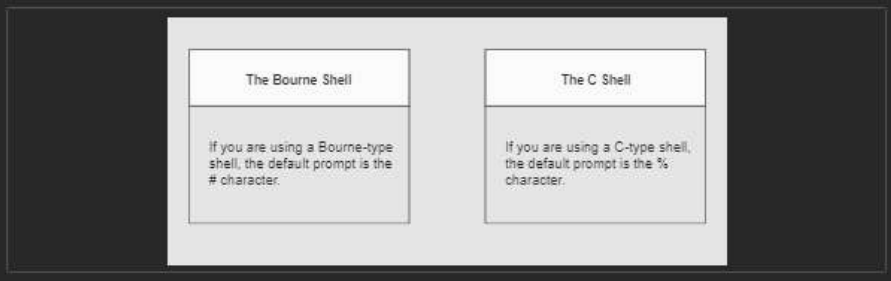
The terminal window in our computer contains a shell that allows you to process information, store or retrieve data and much more by interacting with a computer via entering commands. For example,

- Retrieving a list of files or directories
- Looking up today's date and time
- Getting current directory location
- Making, copying or deleting files
- Sorting files

And much more complicated tasks!

Types of Shell

In UNIX, there are two major types of shell:



Why Shell Scripting?

Just as you can type and execute commands on a Shell terminal, you can put a set of commands in a plain text file with the extension of `.sh`. These files are shell scripts which can be executed like normal `GNU` or `Unix`

- A Unix shell is both, a command interpreter and a programming language. As a *command interpreter*, it provides the user interface to the substantial set of GNU utilities. The *programming language features* allow these utilities to be combined
- Shell scripting allows users to automate their tasks. Script files containing commands can be executed as commands themselves. These customized commands have the same status as system commands in directories like `/bin`. Hence, the users can create their own customized environments to carry out tasks
- Shell scripts can run in an interactive and non-interactive mode
- A shell allows synchronous and asynchronous execution of `GNU` commands
- Shell scripting allows rapid prototyping
- Shell scripting reduces extensive, complex, and repetitive sequences of commands into a simple command

Typical Examples of Shell Programming

- Allows to create user accounts
- Writing application startup scripts, especially unattended applications
- Find out what processes are eating up your system resources
- Find out available and free memory
- Find out all logged in users and what they are doing
- Writing system boot scripts
- User administration as per your security policies
- Find out information about local or remote servers
- Creating application package installation tools
- Automation of customized processes

Introduction to Linux

Delve into all the basic concepts of Linux Shell.

Linux is a free open-source operating system that works just like Windows and Mac OS X. It is a `Unix` clone, multi-tasking and a multi-user operating system that was developed by *Linus Torvalds* with the collaboration of the developers around the world.

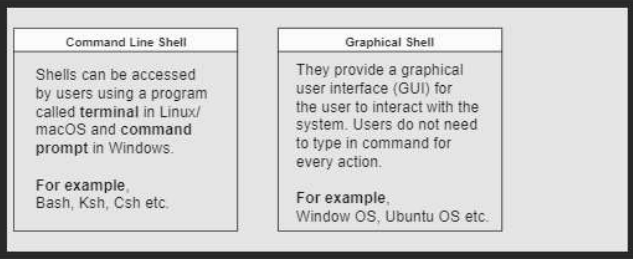
Linux - A Kernel, not an Operating System#

Strictly speaking, Linux is a kernel, not an entire operating system. A kernel provides access to computer hardware and a controlled access to system resources such as:

- Security and firewall
- GNU libraries and utilities
- Other management and installation scripts
- Logged in users
- Running and loading programs to memory
- Networking systems
- Files and data
- Process management
- Device management
- I/O management

Shell Classification

- Command line shell
- Graphical shell



Introduction to Linux Shell

> A shell is an environment provided for the user to interact with the machine.

Shell is not a part of Linux kernel, but it uses the kernel to execute programs, create files etc. Several command line shells are available for Linux including:

- **BASH (Bourne-Again SHell)** - It's the default open source shell on many Linux distributions today
- **CSH (C SHell)** - The C shell's syntax and usage are very similar to the C programming language
- **KSH (Korn SHell)** - It is a high-level interactive command language
- **TCSH** - It is an enhanced but completely compatible version of the Berkeley UNIX C shell (CSH)
- **ZSH** - It is an interactive shell that incorporates features of other Linux shells including some unique ones too
- **Fish (Friendly Interactive Shell)** - It is a smart and user-friendly interactive shell along with some other features

Getting Meta Information in Bash

Explore the commands used to retrieve details about Shell and meta information about commands.

Finding Out Your Default Shell

To find out what your default interpreter is, type this in the command window:

```
ps $$
```

The following sample output shows that we are using Bash Shell:

```
PID   TT  STAT      TIME COMMAND
 4755 s000  S      0:00.02 -bash
```

Now, to find out the *full execution* path of your shell interpreter, you can type `which bash`

```
which shell
/bin/bash
```

Similarly, you can use or `echo $SHELL` for this same purpose:

```
echo $SHELL
/bin/bash
```

Changing Your Default Shell

If your default shell is some other than **bash**, you can conveniently change your shell to bash by entering:

```
chsh -s /bin/bash
```

Getting Command Information in Linux

You can view the official documentation by typing `man` *command* or `info` *command* in the terminal. For example, to open the *manpage* of `cal` command, we use:

```
man cal
```

...and to open the documentation of `date` command, we will run this:

```
info date
```

What is Bash?

From here starts a detailed series of Bash content, sit tight!

Introduction to Bash

Bash is the default Linux shell, a command line interpreter developed by the **GNU** project. It is the free version of the *Bourne Shell*. The name is an acronym for the ***'Bourne-Again Shell'***.

Why Bash?

Bash is an evolved and refined command language which enables you to interact with your operating system. You can type commands for Bash to interpret and execute them.

You might be having a question that why do I have to switch over to the command line from GUI (Graphical User Interface). GUI is rather much easy to use. But I will have to suggest you turn your hand towards the command line for some tasks like file handling, administrative problems, and data manipulation, etc. Inherently, you should use the best tool to perform a certain task.

Common Use Cases of Bash

Following are some of the common applications of Bash:

- File manipulation
- Program execution
- Creating your power tools/utilities
- Automating command input or entry
- Customizing administrative tasks
- Creating simple applications
- Creating customized power utilities
- Printing text

When not to use Bash?

- Need direct access to system hardware or external peripherals
- Need data structures, such as linked lists, graphs or trees
- Need to generate or manipulate graphics or GUIs
- Need native support for multi-dimensional arrays
- Need port or socket I/O
- Extensive file operations (Bash is limited to serial file access, and that only in a clumsy and inefficient line-by-line fashion)
- Where cross-platform portability is required
- Resource-intensive tasks, especially where speed is an important factor (sorting, recursion, hashing, etc.)

In all these complex areas, it is suggested to use a multifunctional scripting language such as *Perl, Python, Ruby* or some high-level programming language like *C, C++, Java*, etc.

Introduction to Command Line

Here, you will briefly cover command line and will discuss a few special characters/operators which come in really handy while using the terminal.

We'll cover the following

What is the Command Line?#

> A *command-line interface* (CLI) or a command line interpreter or shell is simply a mean for the user to interact with the system in the form of progressive and sequential commands.

A useful way for the users to conduct various actions like viewing, handling, and manipulating files on the computer is *Graphical-User-Interface* or GUI. However, CLI is the primary approach to interact with the computer.

CLI has essential *advantages*, especially for meticulous computer users. It lets you more control over your system by allowing you to run system commands and automate your various tasks. Similarly, you can add modifiers to define exactly how you want your program to execute.

Types#

"Shells" have evolved through quite a many adaptations. In *OS X* and *Linux*, most commonly used shell is `bash`, whereas in *Windows* has `MS-DOS` based CLI.

Types#

"Shells" have evolved through quite a many adaptations. In *OS X* and *Linux*, most commonly used shell is `bash`, whereas in *Windows* has `MS-DOS` based CLI.

Open Your Command Line Interface

- In Linux:

Press `Ctrl+Alt+T`

- In OS X:

`Applications → Utilities → Terminal`

- In Windows:

Press `Win-R`, type `cmd`, press `Enter`

`Start → Program Files → Accessories → Command Prompt`

Observe the Prompt

The command line prompts in different ways for different OS environments.

- In Windows: `>`
- In Linux or OS X: `$`

Embark on with your First Command

So, you've made it this far. Hopefully, you might have grasped a basic understanding of shell. Now start off with your first command:

```
1 whoami
```

This command returns the username of the owner of current login session. So, your computer just printed your name, right? Wow!

Basic Command Line Editing

You can use the following key combinations to edit and recall commands:

- **Esc** + **T**: Swap the last two words before the cursor
- **Ctrl** + **H**: Delete the letter starting at the cursor
- **Ctrl** + **W**: Delete the word starting at the cursor
- **TAB**: Auto-complete files, directory, command names and much more
- **Ctrl** + **R**: To see the command history.
- **Ctrl** + **U**: Clear the line
- **Ctrl** + **C**: Cancel currently running commands.
- **Ctrl** + **L**: Clear the screen
- **Ctrl** + **T**: Swap the last two characters before the cursor

Special Characters in Bash

Each special character, in Bash, holds a unique meaning. Let's look at this table to find out the meaning of each character:

Characters	Description
/	Directory separator, used to separate a string of directory names. **Example:** `/home/projects/file`
\	Escape character. If you want to reference a special character, you must "escape" it with a backslash first. **Example:** `\n` means newline; `\v` means vertical tab; `\r` means return
#	Lines starting with `#` will not be executed. These lines are comments
.	Current directory. When its the first character in a filename, it can also "hide" files
..	Returns the parent directory
~	Returns user's home directory
~+	Returns the current working directory. It corresponds to the `$PWD` internal variable
~-	Returns the previous working directory. It corresponds to the `$OLDPWD` internal variable

*	Represents 0 or more characters in a filename, or by itself, it matches all files in a directory. **Example:** `file*2019` can return: `file2019`, `file_comp2019`, `fileMay2019`
[]	Can be used to represent a range of values, e.g. [0-9], [A-Z], etc. **Example:** `file[3-5].txt` represents `file3.txt`, `file4.txt`, `file5.txt`
|	Known as "pipe". It redirects the output of the previous command into the input of the next command. **Example:** `ls | less`
<	It redirects a file as an input to a program. **Example:** `more < file.txt`
>	In **script name >filename** it will redirect the output of "script name" to "file filename". Overwrite filename if it already exists. **Example:** `ls > file.txt`
>>	Redirect and append the output of the command to the end of the file. **Example:** `echo "To the end of file" >> file.txt`
&	Execute a job in the background and immediately get your shell back. **Example:** `sleep 10 &`
&&	"AND logical operator". It returns (success) only if both the linked test conditions are true. It would run the second command only if the first one ran without errors. **Example:** `let "num = (( 0 && 1 ))"; cd/comp/projs && less messages`
;	"Command separator". Allows you to execute multiple commands in a single line. **Example:** `cd/comp/projs ; less messages`
?	This character serves as a single character in a filename. **Example:** `file?.txt` can represent `file1.txt`, `file2.txt`, `file3.txt`

Exiting the Command Line

You can now safely exit from the Shell by writing:

```
1 exit
```

Commands & Arguments (Optional)

A short introduction to commands, arguments, and options.

If you're a beginner, then it is strongly recommended that you read this section to grasp the basic concepts of commands, arguments, and options. If you are somehow familiar with these concepts already, then you can skip this lesson and jump to the next lesson where you can test your command line skills by using interactive code widgets. Good luck!

Command Syntax#

The general syntax followed by any Bash command is:

```
command_name [-option(s)] [argument(s)]
```

Let's divide this syntax and understand what each of the terms mean:

Command Name:

A command name is a unique word which is used to indicate to the system what action needs to be performed. Each command has its own set of arguments and options to narrow down the functionality even more. For example, `ls` is a very commonly used command.

Argument:

Any Bash command takes a list of arguments to indicate to the system which objects to look for while performing the action. An argument could be a string, set of string or a token passed to the command. For example, the command `ls` can take a directory's path as an argument.

```
ls /home/user/Downloads/Music
```

Options:

An option is also referred as the “mode” of the command. It controls the behavior of the command. It is a single character carry that carries a unique meaning. Most of the commands run without the option. Some commands can also take multiple sets of options simultaneously. For example, the command `ls` can take `-a` as an option which generally means “all” and is used to show hidden files.

```
ls -a
```

Important Points:#

1. Some commands might not take any arguments or options such as `pwd`
2. Any option is written with a hyphen (-)
3. A double hyphen (–) is used to show the end of options if more than one options are used
4. The order of argument matters sometimes
5. In some commands, we can also pass flags as arguments. It simply carries a true or false value which could be used as an indication to perform something or not.

Finding System Date and Time

This lesson will guide you through getting system date and time in your desired format.

We'll cover the following...

date#

Definition:

The `date` command is used to print out or change system time and date.

Syntax:

```
date [+FORMAT]
```

Format:

Format Sequence	Description
%a	The abbreviated weekday name (e.g., Mon).
%A	The full weekday name (e.g., Monday).
%b	The abbreviated month name (e.g., Feb).
%B	The full month name (e.g., February).
%c	The date and time (e.g., Fri Feb 6 21:45:39 2018).
%d	Day of month (e.g., 04).
%D	Date; same as %m/%d/%y.
%F	Full date; same as %Y-%m-%d.

Example:

- To get system time and date:

```
date
```

```
1 date
```

Run Save Reset

- To set system time and date to **July 01, 2018, 03:15 PM**:

```
date -s "07/01/2018 03:15:00"
```

- To get date in the format *year-month-day hour:minute:second* from `date`:

```
date '+%Y-%m-%d %H:%M:%S'
```

```
1 date '+%Y-%m-%d %H:%M:%S'
```

Run Save Reset

- To get the output in the following format:

```
date '+DATE: %m/%d/%y%nTIME: %H:%M:%S'
```

Other Commonly Used Bash Commands

Get to learn some useful and important bash commands that are necessary in order to interact with your machine efficiently and get your day-to-day tasks done.

1. echo

Definition:

`echo` is the built-in command in bash and C-shells that simply prints its arguments on the console or terminal.

Syntax:

```
echo [option(s)] [string(s)]
```

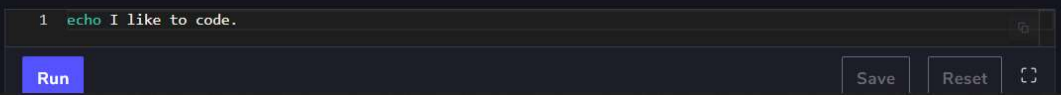
Options:

Options	Meanings
-n	Do not output a trailing newline.
-e	Enable interpretation of backslash escape sequences.
-E	Disable interpretation of backslash escape sequences.
–help	Display a help message and exit.

Example:

- To print a statement “I like to code.” on the console

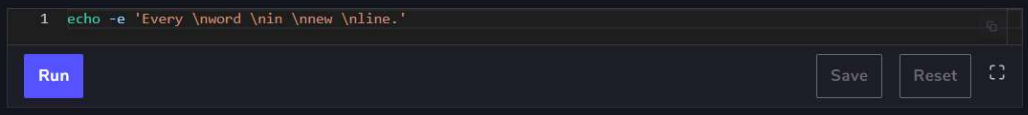
```
echo I like to code.
```



- To print a statement having words in each consecutive line

```
echo -e 'Every \nword \nin \nnew \nline.'
```

Here, `\n` represents a “new line”, and `-e` is enabling for the command line to interpret the backslash.



2. clear#

Definition:#

The clear command is used to remove all previous commands from *console*. It neither accepts “options” nor “arguments” (input files). After it is executed, all there is left on the command line is the *command prompt* on the upper left corner.

Syntax:#

```
clear
```

3. sleep#

Definition:

Sleep command pauses for some time as specified by the **NUMBER**

Syntax:

```
sleep NUMBER[SUFFIX]
```

The `SUFFIX` may be:

- *s* for seconds (the default)
- *m* for minutes
- *h* for hours
- *d* for days

Example:

To sleep for 4 hours:

```
sleep 4h
```

To sleep for 5 days:

```
sleep 5d
```

Pathnames in Bash

Get yourself acquainted with the necessary concepts related to paths and their types.

Before hopping on the Navigation commands, it is important to cover the terms and concepts necessary to understand those commands. This lesson, again, is optional if you are already familiar with the terms such as Relative and Absolute path and know what references are. If you have even a slightest of doubt about any of these concepts, then it is suggested that you give this lesson a thorough read. It will clear all your doubts and confusions and will help you understand it with simple examples. So let's get to it one by one:

What is a Path?

A path can be a location to any folder or subfolder in your file system. Any path starts with the slash symbol (/) and contains names of different sub-folders in the hierarchy. You have to add a slash after the name of every sub-folder that you mention in the path.

When you open your terminal, it by default points to your "home" folder which is usually referred as "current directory". If you want to run any file by its name only, you need to be the first present in that directory where that file is stored, by changing your current directory. Otherwise, you can only run that file by specifying the complete path.

Examples:

- /home
- /usr/bin/local/app

- /home
- /usr/bin/local/app

There are two types of paths:

- Absolute Path
- Relative Path

Absolute Path

The pathname in which you specify the complete path from root to where the file is stored is called *Absolute Path*. This pathname is a combination of folder names and slashes, along with the file name and its extension. Every object stored in your filesystem will have a unique pathname. Here are few examples to give a better understanding:

- /home/downloads/music.my_song.mp3
- /home/Documents/file.txt

> **Note:**
>
> The first `/` in a pathname means the root directory and each `/` that we add in the path desends to the sub-level in your file system.

Relative Path

A relative path starts from your present working directory (pwd) also called “current directory” instead of the root directory. So instead of specifying the whole path from root to the file, you just specify it from your current directory. This technique comes in handy when the files resides in one of the child directories of your current directory. For example, if there’s a hierarchy of directories in your filesystem where **Download** is your current directory and you need to move to its child directory

> **Note:**
>
> In Unix, a `.` is used to represent the current directory at which you’re standing and `..` means *parent directory*.

Create a Directory in Bash

This lesson will enable you to create directories through command line interface.

mkdir#

Definition:

`mkdir` is one of the most important directory manipulation commands. It's short for 'make directory' and as the name suggests, it is used for creating new directories within the current directory. You can create any number of directories with this command. It takes the directory name as an argument.

Syntax:

```
mkdir [options][dir_name]
```

Options:

`mkdir` in terminal can take two options:

Options:

`mkdir` in terminal can take two options:

Option	Meaning
-p	Means Parent or Path. It is used when we want to create a directory within a directory that doesn't already exist
-m	Means Mode. It is used to specify and control permission modes

Example:

```
mkdir my_dir_1 my_dir_2 my_dir_3
```

```
1 mkdir my_dir_1 my_dir_2 my_dir_3
2 ls
```

Run Save Reset

Remove a Directory in Bash

Here, you will learn how to remove directories in bash.

rmdir#

Definition:

`rmdir` is the opposite of `mkdir`. It is used to remove the directory name given in the argument. This command is mainly used to remove empty directories. Just like you can make muliple directories at the same time, you can also delete multiple directories simultaneously by passing their names as arguments.

Syntax:

```
rmdir [option] [dir_name]
```

Options:

`rmdir` in terminal can take two options:

Option	Meaning
-p	This option tells rmdir to remove nested directories if they become empty after deleting previous directory
-v or verbose	Verbose mode outputs after every directory operation that you perform

Example:

```
rmdir my_dir_1 my_dir_2 my_dir_3
```

```
1  echo "Current Directory Structure"
2  ls
3
4  rmdir my_dir_1
5  echo "Updated Directory Structure"
6  ls
```

Run Save Reset

Create a File in Bash

Pick up the different ways to create files via terminal!

This chapter will give you a brief introduction about File Manipulation commands which you must know before moving on to Shell Programming. File Manipulation commands lets you perform necessary operations on your files through terminal. Some of the most powerful commands are given below:

touch#

Definition:#

`touch` command is considered one of the easiest ways to create new files because it does not overwrite the existing files unlike `cp`, `rm` etc. It is also used to change the timestamps of files. A timestamp can be of three types: Access Date & Time, Modification Date & Time and the Time & Date when meta information associated with file was changed. By default, the behavior of this command is set to create empty files only. You can use different options to change file's timestamp information.

Syntax:#

```
touch [option] [file_name(s)]
```

Options:#

Several options can be used together to simultaneously change the timestamp information associated to the file.

Option	Meaning
-am	A is for Access Time and M is for Modification Time. You can change both together by using this option
-r	Means *reference*. It makes a reference to another file's timestamps and use them for your file
-B	Means *Back*. This options changes time by going a few seconds back, specified by the user
-F	Means *Forward*. This options changes time by going a few seconds forward, specified by the user
-d or -t	These two options are used if the user wants to add his/her own access time in a specific format

Example:

1. Create a file named “my_file”

```
touch my_file
```

```
1 echo "Current Directory Files:"
2 ls
3
4 touch my_file
5
6 echo "Updated Directory Files:"
7 ls
```

Run Save Reset

2. Change the Access Time of file named “my_file”. Here, `ls -lu` displays the access time of all files.

```
touch -a my_file
```

```
1 echo "Current Directory Files:"
2 ls
3
4 touch -a my_file
5
6 echo "Updated Directory Files:"
7 ls
8 ls -lu
```

Run Save Reset

Remove a File in Bash

Here, you will learn how to delete files in a shell.

rm

Definition:

`rm` is used to delete files and directories. By default, it is only set to remove files and not directories for safety. It basically unlinks the file name with the data stored in so that it cannot be accessed anymore. Before deleting any file, you must have permissions to delete it. You can delete multiple files at once. This command is almost similar to `rmdir`. The only difference is that `rmdir` is used to delete empty directories only. You can also delete symbolic links to a file, in this way, only link is removed but the file remains unaffected.

Syntax:

```
rm [options] [file_name(s)]
```

Options:

To indicate `rm` the end of options, `--` double dashes are used. This is very useful when a directory's name starts with `-`

Option	Meaning
-f	Means *force*. This option tells `rm` to force delete all the specified files without showing any message
-i	Means *interactive*. This option prompts the user for confirmation before deleting any file
-r or -R	Means *recursive*. It is used to recursively delete diectories by first emptying them and then removing them one by one

Example:

1. Delete multiple files at once

`rm file_1 file_2 file_3`

```
touch file_1 file_2 file_3 file_4 #Create 4 files
echo "Current Directory Files:"
ls

rm file_1 file_2 file_3
echo "Updated Directory Files:"
ls

```

Run Save Reset

2. Delete a directory named "my_dir"

`rm -r my_dir`

```
mkdir my_dir my_dir_2
echo "Current Directory Structure:"
ls

rm -r my_dir
echo "Updated Directory Structure:"
ls
```

Run Save Reset

Open/Display Content of a File

This lesson will illustrate various ways to display content and word count of files.

wc#

Definition:

`wc` (word count) is used to get word count, newline count, byte and characters count in the files specified in its input.

Syntax:

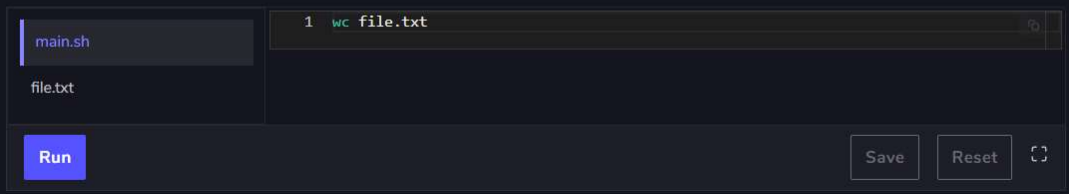
```
wc [option] [file]
```

Options:

Option	Description
-m	Print the character counts.
-c	Print the byte counts.
-l	Print the newline counts.
-w	Print the word counts.
-L	Print the length of the longest line.

Examples:

Execute the following example. Here, **6** is the number of lines, **9** is the number of words, and **65** is the number of characters.



cat

Definition:

`cat` commands — short for concatenation — can be used for multiple purpose. The three main uses of `cat` are: concatenation, read file and new file creation. Cat is considered one of the most reliable methods to open a file in a read mode and print its content.

Syntax:

Concatenation:

```
cat [option] [file_name(s)]
```

File Reading:

```
1. cat [file_name(s)] // To print contents of file
```

```
2. cat [file_name] > [file_name] // To redirect content to another file
```

```
3. cat [file_name] | less // To filter the content to be displayed with piping
```

File Creation:

```
1. cat > [new_file_name]  // To create new file or overwrite if file already exists
```

```
2. cat >> [existing_file_name] // To preserve previous file if it already exists by appending any new te
xt
```

Operators:

Operator	Meaning
`>`	*Redirection Operator*. Redirects the contents of one file to another
`>>`	*Append Operator*. Appends the content of one file at the end of the other file. Used to prevent over-writting issues.
`|`	*Piping Operator*. Pipes the content of a file if its too large to display

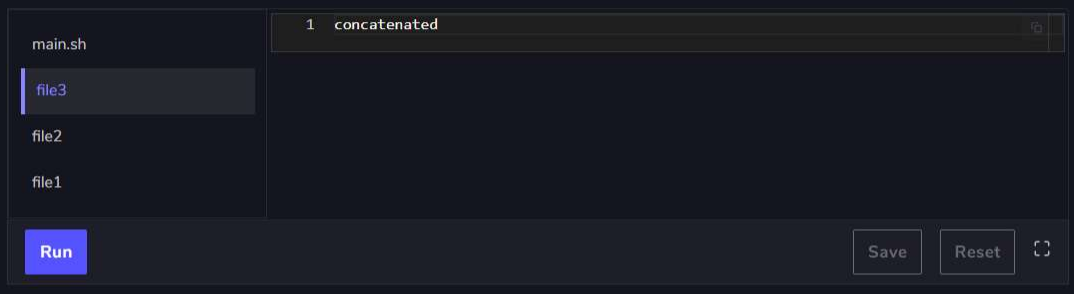
Options:

Option	Meaning
-n	Display the contents of file with line numbers
-E	Concatenate '$' at the end of each line of file
-T	Replaces `tab` as `^I`
-v	To show non-printable characters on command line
-A	Combination of v,E and T

Examples:

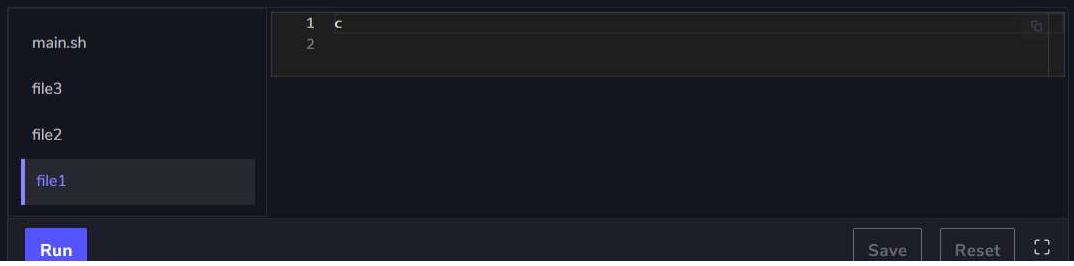
1. To concatenate the content of three files and then redirect the output to a fourth file:

```
cat file1 file2 file3 > file4
```



2. To pipe the content of three files given as arguments and then sort them in alphabetical order in a new file

```
cat file1 file2 file3 | sort > file4
```



Piping?

You might have heard this term on various platforms. Piping, in general, is a very broad topic itself but we would limit our discussion here by only considering its use in file manipulation domain. It's a common technique used in Linux to redirect or chain the content to another destination. The destination could be another program, another file or even an output device such as printer. This is oftenly used when the content of a file are to large so we are bound to pipe the content and direct them somewhere else.

Move Files in a Directory

Move files in a directory easily in Linux via "mv" command.

We'll cover the following...

mv

Definition:

The `mv` command is one the frequently used commands which is used to move files and directories. We can also use this command to rename files and directories. File names or directory names are given as an argument. If the files given as argument are present in the same directory then you can only *rename* them, not move them. The source file name would be renamed as destination file name. You can move multiple files/directories simulatenously using `mv` command.

Syntax:

```
mv [options] [source] [destination]
```

Options:

Option	Meaning
-i	Means *intercative*. This option is only used to warn the user about overwrite issues
-b	Used to make backup copies of files
-v	Means *verbose*. It displays information about each file being processed.
- -help	Gives detailed information about `mv` command
- -version	This option is to check the version of the installed program

Example:

1. Rename a file and keep it in the same directory `mv my_file_1 my_file_2`

```
touch my_file_1
mv my_file_1 my_file_2
ls
```

Run Save Reset

2. Move all files, directories from current directory to specified directory. Hint: `*` means all.

`mv * /home/sub-dir/new/`

```
touch file1 file2 file3 #Make 3 files in current directory
mv * ../new #Move all files in current directory to another directory
cd ../new #Switch to the new directory
ls #Show all files present in the new directory
```

Run Save Reset

3. Move a file from sub-sub directory to the user's home directory. Hint: User's home directory can be accessed using `~` sybmol.

`mv dir/dir/file4 ~`

Copying Files in Bash

You will learn how to copy files via command line in this lesson.

We'll cover the following...

cp

Definition:

This command is one of the most frequently used commands. It is used to copy files and directories. By default, `cp` copies files but not directories. Always be careful while choosing name for new files because the content might be lost if you use a file name that already exists. If you're placing the copied version in the same directory then it must have a different name than the original name.

Syntax:

```
cp [option] [file_name] new_file_name
```

Options:

Option	Meaning
-r or -R	Means *Recursive*. It is used to copy directories including all its content
-i	This option is only used to warn the user about overwrite issues
-b	Used to make backup copies
-f	Means *Force*. This option is used to force open the destination files
-u	Means *Update*. As the name suggests, it only updates the file if any changes are made in the file
-x	This option is to indicate `cp` to stay on the same file system
-s	By using this option you can only make references rather than deep copying everything

Example:

1. Copy three files in a different directory simultaneously with the same names

```
cp file1 file2 file3 my_directory
```

2. Copy all files from current directory to “my_directory” with only .txt extension

```
cp *.txt my_directory
```

3. Copy all files from one directory to another directory using star *wildcard*

```
cp my_dir_1/* my_dir_2
```

> ### Wildcards?
>
> The above example uses a wildcard character i.e. `*` to run the command. Wildcards are commonly used in Shell commands. Some of the most frequently used are: `?`, `#`, `[]` etc. The star wildcard simply means “any” or “every”. Thus, the star wildcard, wherever used, would mean to return “any” of the possible result or object.

Zipping a File in Bash

Get to know the commands to zip files in bash.

We'll cover the following...

tar#

Definition:#

Tar — short for *tape archive* is used to convert files in *tar* format and *archive* them in a file. An archive format is widely used to store multiple files and transfer them to another system. They can be extracted, once recieved on the second end. We use Tar command for all these purposes.

Syntax:#

```
tar [option(s)] [archive_file_name] [file_name(s)]
```

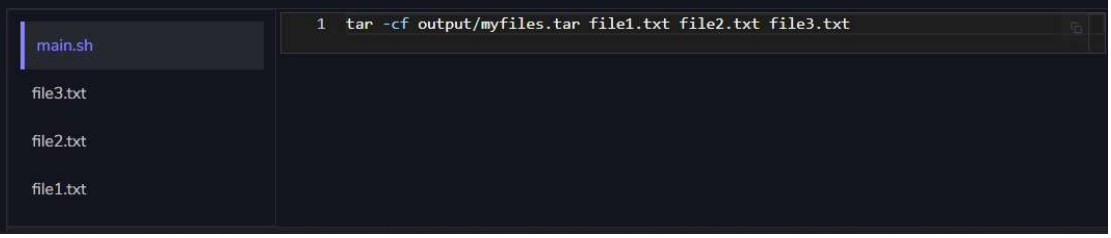
Options:#

The `tar` command is one of those rare commands which requires to specify at least one option in order to work properly. Here are a few options commonly used with `tar` command, you can use them individually or combine them in a single tar command.

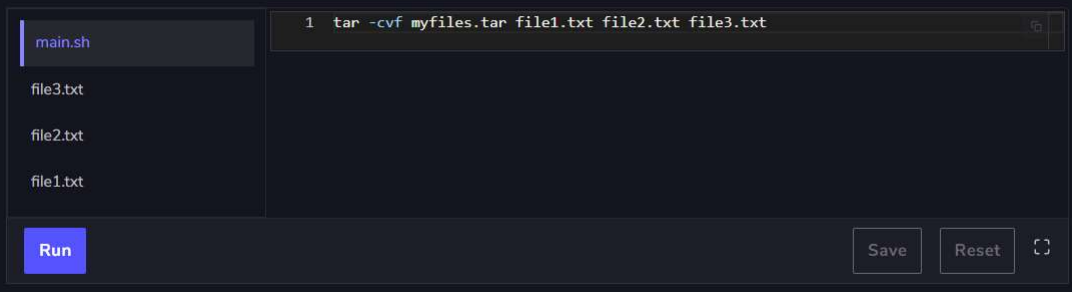
Option	Meaning
-c	Creates an archive
-x	Extracts an archive
-f	Create archive with the given name
-t	Display the files in archive
-z	Creates archive with gzip
-r	Update/Add the archive file with new files
-v	Verbose

Example:

1. Archive three files into a single tar file and name it “my_files.tar”



2. Archive the files just as above and display a list of files present in the archive file



gzip#

Definition:

Gzip is another file format which could be used to group the files together and compress them in order to save space. Zip and Gzip both file formats are widely used for this purpose. The extension for gzip files is `.gz`

Syntax:

```
gzip [option(s)] [file_name(s)]
```

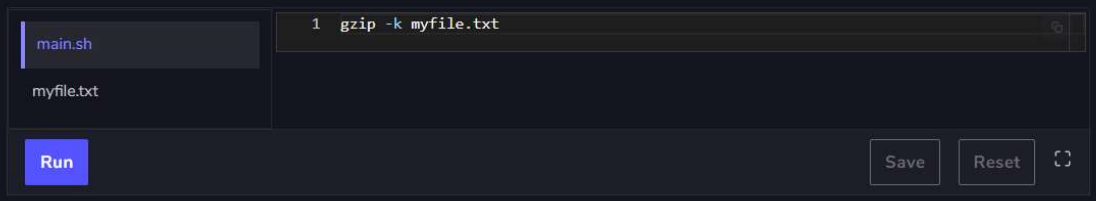
Options:

Option	Meaning
-k	Compresses the file in its original form
-d	Decompresses the gzip file
-r	Compresses the file in a folderwise manner and places them in their respective gzip files
0-9	Used to set compression levels
-v	Verbose. Displays information about compression

Example:

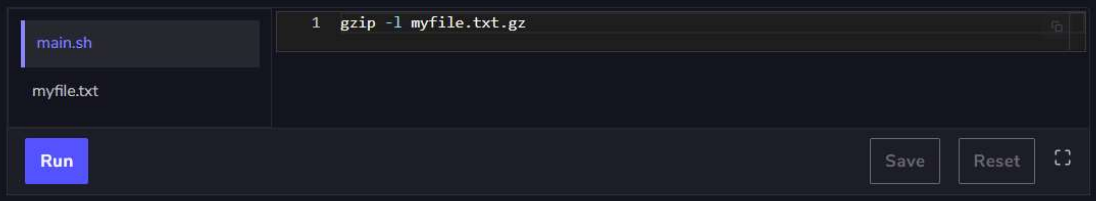
1. Compress a file using gzip but keep it in original form

```
gzip -k myfile.txt
```



2. Show the compression details of a file.

```
gzip -l myfile.gz
```



What are Permissions?

This lesson will tell you everything you need to know about permissions in GNU/Linux.

In Linux, each file or directory follows certain set of rules and restrictions regarding who can access the file and change its content. These rules or *access modes* are commonly referred as *permissions* in Bash. Permissions are sometimes very useful as it gives us the benefit of hiding certain things from other users or make them non-editable for them.

Types of Permissions:

Based on the kind of actions a user can perform on a file, the three basic type of permissions are:

- Read
- Write
- Execute

Symbolical Permissions:

Mode	Meaning
r	Read the file
w	Write/Delete the file
x	Execute the file or search the directory

Numerical Permissions:

Given below is a list of numerical modes which can be changed for the owner, group users, and everyone else. A symbolical equivalent is also mentioned in the next column.

Mode	Meaning
0	000 -> `---` -> None
1	001 -> `--x` -> Execute Only
2	010 -> `-w-` -> Write only
3	011 -> `-wx` -> Write and Execute
4	100 -> `r--` -> Read only
5	101 -> `r-x` -> Read and Execute
6	110 -> `rw-` -> Read and Write
7	111 -> `rwx` -> Read, Write & Execute

Set/Remove Permissions in Bash

This lesson will be covering the command that assists in setting or removing permissions.

chmod#

Definition:

`chmod` command is short for “change mode”, is typically used to change the read/write permissions of files and directories. A mode can either be represented symbolically with unique alphabets or by the permutations of three numbers (octal numbers) as we discussed in the previous lesson. We can also add references in the command to indicate the users on which the specified permissions are being applied. Given below is the syntax to `chmod` command.

Syntax:

```
chmod [references] [options] [mode] [file_name]
```

Operators

Operator	Meaning
+	Adds the specified permissions on the files
-	Removes the specified perfmissions on the files
=	Add ONLY specified permissions on the file and remove the remaining permissions/modes

References

References are basically used to indicate the users on which the permissions are applied.

Reference	Meaning
u	User who owns the file(s)
g	Users in the file group
o	Users which are neither owner of the file nor in the file group
a	All users. Same as `ugo`

Options

Option	Meaning
-f	suppress error messages
-v	outputs the file being processed
-c	to output warnings before making any changes
-R	Means *Recursive*. This option is used to change directories recursively

Example:

1. You can set permissions by writing combination of three digit numbers where first digit is to set permission for the owner, second digit is to set permissions for the file group members, and third is to set permissions for other users. OR you can explicity write it as:

```
chmod u=rw,g=wx,o=r my_file.c
```

The numerical equivalent for this same command where owner can read and write the file, group members and write and execute the file and other users can only read the file:

```
chmod 634 my_file.c
```

2. To remove the permission of “execute” for a file for everyone:

```
chmod a-x my_file.txt
```

3. Allow other users to execute and read the file

```
chmod o+rx my_file.txt
```

Searching in Bash

How to search files and directories based on various attributes, using different methods. Moreover, this lesson encompasses the ways to search for files based on their contents and patterns.

locate#

Definition:#

The command `locate` is a quick way to search for the locations of files and directories.

Syntax:#

```
locate [options] name(s)
```

Options:

Option	Description
-q	To suppress error messages, such as those that might be returned in the event that the user does not have permission to access designated files or directories.
-n	This option followed by an integer limits the results to a specific number.
-i	To perform a case-insensitive search.
-V	To show which version of locate is used, including whether it is locate or slocate.

Example:

- To search for all files named 'file1' and all directories named 'dir1', for which the user had access permission:

```
locate file1 dir1
```

- To display all the files in the system with **".jpeg"** extension:

```
locate "*.jpeg"
```

- In the above example, in order to display only **10** results:

```
locate -n 10 "*.jpeg"
```

find#

Definition:#

`find` is a very powerful command which is used to search for specific patterns of texts within files and directories.

Syntax:#

```
find [path...] [expression]
```

Tests (filters):#

`find` has the ability to filter which files and folder it “selects”. They are called tests.

Test	Description
-type f	Selects files.
-type d	Selects directories.
-name	True if the base of the file name (the path with the leading directories removed) matches shell pattern pattern.
-iname	Search without regard for text case.
-prune	To ignore a whole directory tree.
-path	This is the exact same as name, except that it doesn’t only apply on the filename, but the whole path.
-not	Return only results that do not match the test case.

Examples:

- Searching for file with a particular name:

To find “img.png” file in `runner/projects`:

```
find runner/projects -name "img.png"
```

```
1 find runner/projects/ -name "img.png"
```

Run Save Reset

- Searching files with specific pattern:

To find all header files in `runner/projects`:

```
find runner/projects -name "*.h"
```

```
1 find runner/projects -name "*.h"
```

Run Save Reset

- Deleting files using find command:

To delete `square.h` file which is in `runner/projects`:

```
1 find runner/projects -name "square.h" -delete
```

Run Save Reset

- Excluding files with specific pattern from the search:

If we want to exclude all `.h` files from the search, and display all other files in the “runner/projects” directory:

```
find runner/projects -not -name "*.h"
```

```
1 find runner/projects -not -name "*.h"
```

Run Save Reset

Difference between `find` and `locate`

locate	find
`locate` uses a prebuilt database i.e. an older version of database which should be regularly updated.	`find` searches in real time and iterates over the filesystem to locate files.
`locate` is much faster.	`find` is relatively slower.
`locate` uses a pattern matched against file names, and searches for only files and directories.	`find` provides more meticulosity, as you can filter files by every attribute of it (size, owner, modification time, permissions etc).

Sorting File Contents in Order

Sorting file contents via command line interface.

sort

Definition:

It is used to sort a program or file that is given in its input.

Syntax:

```
sort [option] filename
```

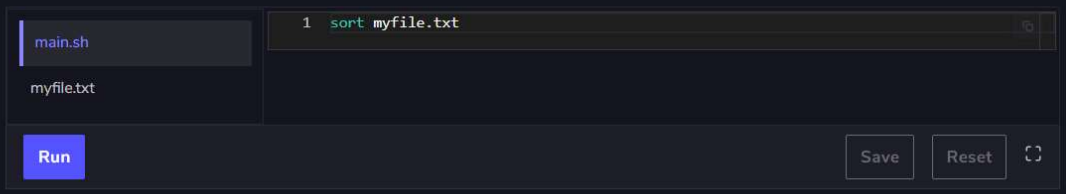
Options:

Option	Description
-o	To write the output to a new file.
-r	To sort in reverse order.
-n	To sort a file numerically.
-nr	To sort a file with numeric data in reverse order.
-k	To sort a table on the basis of any column number.
-c	To check if the file given is already sorted or not.
-u	To sort and remove duplicates

Example:

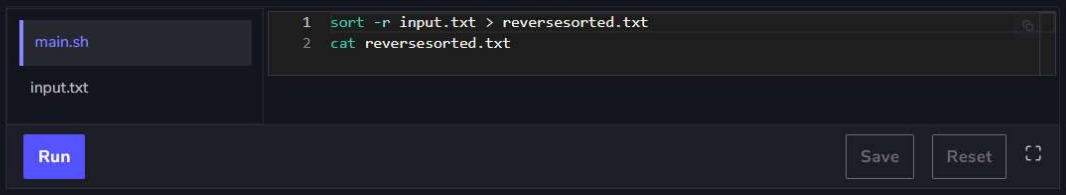
- To simply sort a file:

```
sort myfile.txt
```



- To sort an input file in reverse order and store the output in say, “reversesorted.txt” file:

```
sort -r input.txt > reversesorted.txt
```

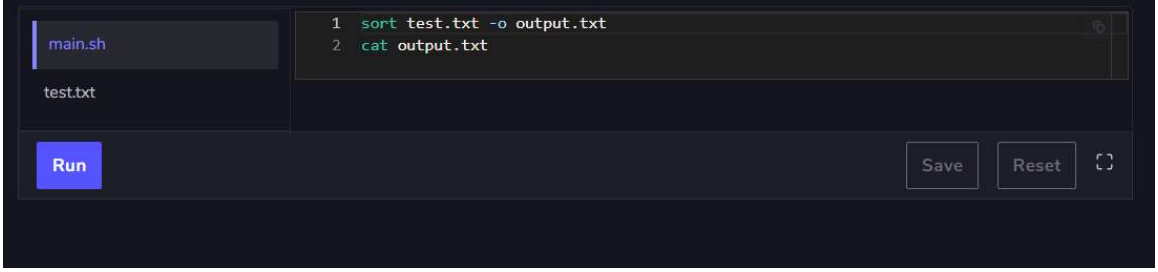


- To sort the contents of “test.txt” and store the results in “output.txt”:

```
sort test.txt -o output.txt
```

- To sort the contents of “test.txt” and store the results in “output.txt”:

```
sort test.txt -o output.txt
```



Viewing Beginning and Ending Contents of Files

Learn the commands to display the beginning and ending contents of any file.

We'll cover the following...

> We can use `head` and `tail` to view beginning and ending contents of file.

head:#

Definition:#

`head`, by default, prints the first 10 lines of each file to standard output. It reads the first few lines of any text given to it as an input and writes them on the display screen.

Syntax:#

```
head [options] [file(s)]
```

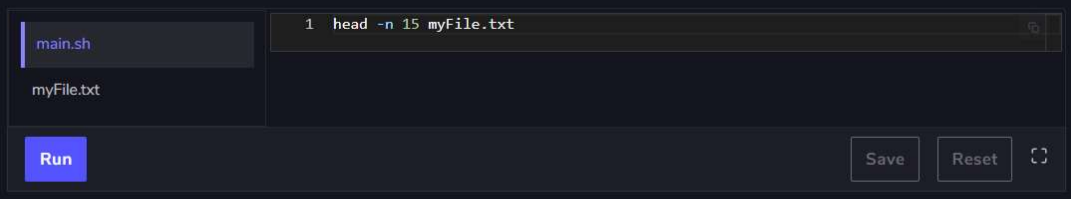
Options:

Option	Description
-n	It can be used followed by an integer representing the number of lines to be displayed.
-c	This option can be used followed by the number of bytes desired.
-q (quiet)	Never print headers identifying file names.
-v (verbose)	Always print headers identifying file names.

Example:

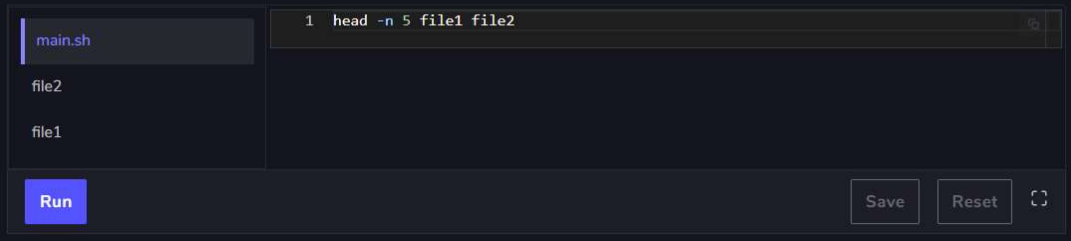
- To display first **15** lines from a file:

```
head -n 15 myFile.txt
```



- To display first **5** lines from both files:

```
head -n 5 file1 file2
```

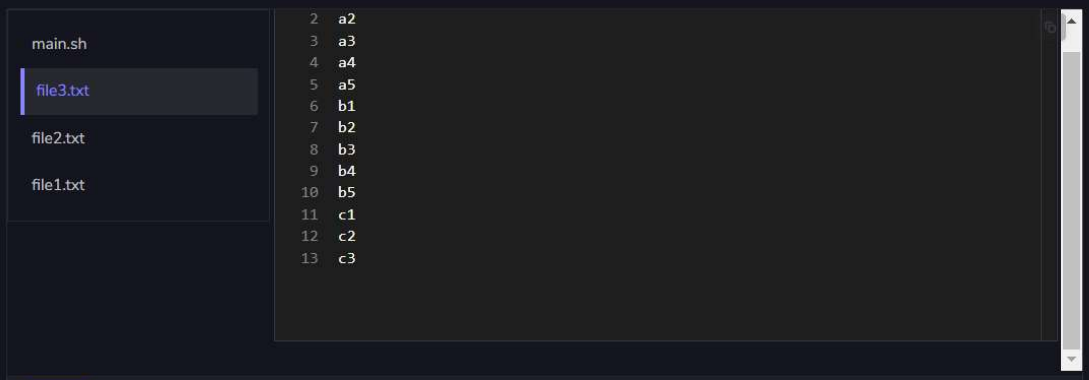


- To display first 10 lines of the three files: “file1.txt”, “file2.txt” and “file3.txt”:

```
head file1.txt file2.txt file3.txt
```

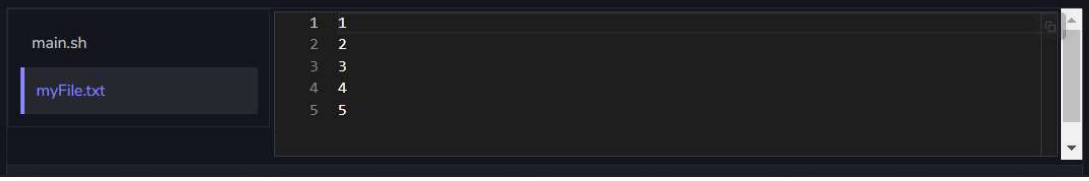
- To display first 10 lines of the three files: “file1.txt”, “file2.txt” and “file3.txt”:

```
head file1.txt file2.txt file3.txt
```



- To display the first 5 bytes of myFile.txt:

```
head -c 5 myFile.txt
```



tail#

Definition:#

`tail` reads the last few lines of any file or text given to it as an input and writes them to the standard output.

Syntax:#

```
tail [options] [file(s)]
```

Options:#

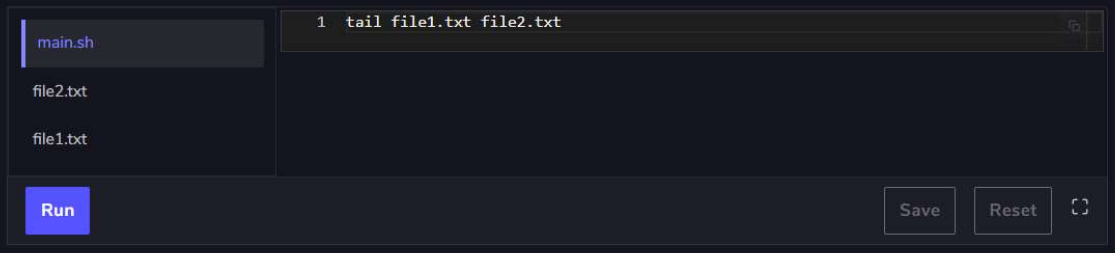
Options:

Option	Description
-n	It is followed by an integer indicating the number of lines that are to be printed.
-c	To print the specific number of bytes. this option precedes the number of bytes.
-q (quiet)	It causes tail to not print the file name before each set of lines and to eliminate the vertical space between each set of lines when there are multiple input sources.
-v (verbose)	It causes tail to print the file name even if there is just a single input file.

Examples:

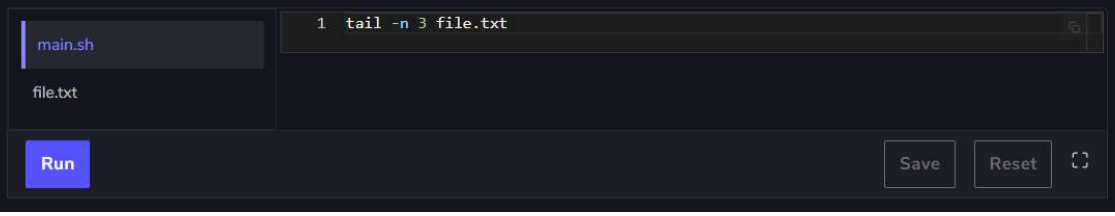
- To print the last 10 lines of the files named “file1.txt” and “file2.txt”:

```
tail file1.txt file2.txt
```



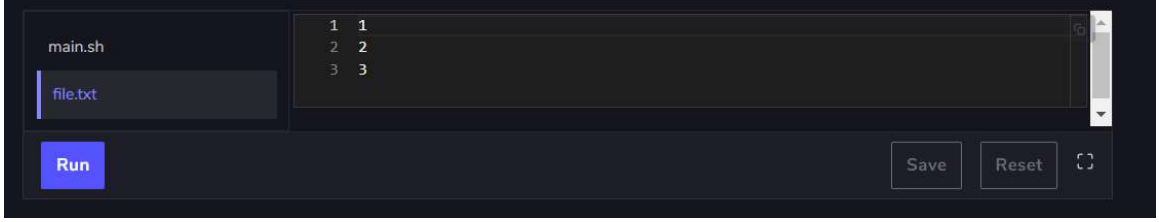
- To display last 3 lines of a file, “file.txt”:

```
tail -n 3 file.txt
```



- To print the last 3 bytes of the file “file.txt”:

```
tail -c 3 file.txt
```



Filtering Repeated Lines Out

Review different ways to search for repeated lines in a text or a file.

uniq#

Definition:#

`uniq` command in bash is a command line utility to filter and view multiple repeated lines.

> This command works on *adjacent comparison lines* so it is often combined with the **sort** command.

It can be used to:

- remove duplicates.
- show only repeated lines.
- show a count of repeated occurrences.
- comparing particular fields and ignoring certain inputs.

Syntax:#

```
uniq [option] [input[output]]
```

Options:

Option	Description
-c	Prefix lines with a number showing how many times they occurred.
-d	Only print duplicated lines.
-u	Only print unique lines.
-z	End lines with 0 byte (NULL), instead of a newline.
-w	Compare no more than **N** characters in lines.
-i	To perform case-insensitive comparisons.
-f	To avoid comparing first **N** fields of a line before determining uniqueness. (***Field*** *is a set of characters delimeted by a white space.*)
-s	To avoid comparing first **N** characters before determining uniqueness.

Examples:

- To display repeated lines:

main.sh

```
1  uniq file.txt
```

file.txt

Run Save Reset

- To ignore first field, containing line numbers from each line while doing comparison, use `-f` option and execute the following snippet. In this way, both lines will be considered alike:

main.sh

```
1  uniq -f 2 file.txt
```

file.txt

Run Save Reset

- Now using **-s** option, we can ignore characters instead of fields while comparing:

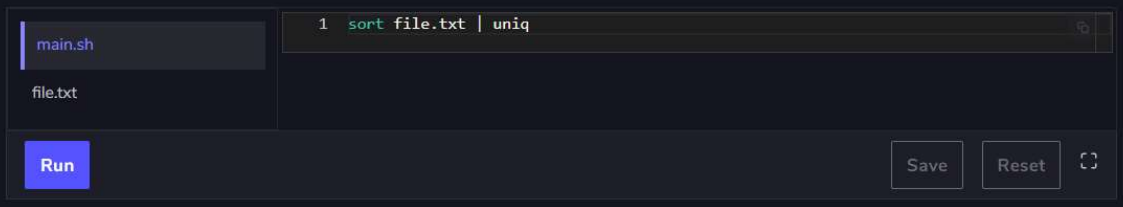
main.sh

```
1  uniq -s 3 file.txt
```

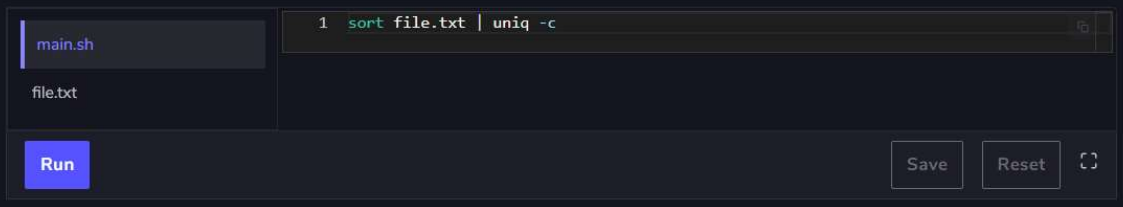
file.txt

Run Save Reset

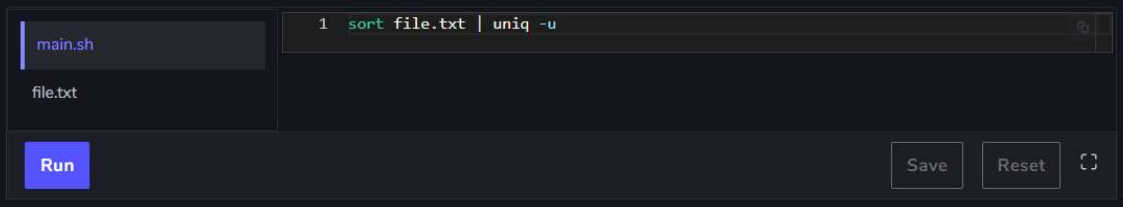
- `uniq` command works on the input that is already sorted. Hence, it is usually combined with the `sort` command to find repeated lines:



- To show a count of the number of times a line occurred, we use the option `-c`:



- To show lines that are not repeated, we use the option `-u`:



Regular Expressions (Regex)

This lesson gives a conceptual summary of regular expressions.

> Regular expressions are a powerful way to filter out specific pieces of information by using various arithematic patterns to describe certain set of strings.

You are probably now familiar with the *wildcards* e.g. ***.txt** to find all text files in some directory. Its regex would be something like this: `^.*\.txt$.`.

Some very important bash commands like `grep` and `egrep` use regex to search for different patterns of texts in inputs and files. Given below is a summary to make you aware of what various regex notations mean.

Regex Pattern Notations:

Regex Operator	Description
?	The preceding item is optional and matched at most once.
*	The preceding item will be matched zero or more times.
+	The preceding item will be matched one or more times.

{n}	The preceding item is matched exactly n times.
{n,}	The preceding item is matched n or more times.
{n,m}	The preceding item is matched at least n times, but not more than m times.
$	Matches the end of the line.
^	Matches the beginning of the line.
()	Allows us to group several characters to behave as one.
|	Its the logical OR operation.
.	A single character.
[agd]	The character is one of those included within the square brackets.
[^agd]	The character is not one of those included within the square brackets.
[a-d]	The dash within the square brackets operates as a range. Here, it means all characters between **a** and **d**, including “a” and “d”.

GREP vs. EGREP vs. FGREP

Search for a variety of text fragments through the most powerful commands of terminal i.e. grep, egrep and fgrep.

We'll cover the following...

grep#

Definition:#

The command `grep` stands for **"global regular expression print"**, and is used to search for specified text patterns in files or program outputs.

Syntax:#

```
grep [option(s)] pattern [file(s)]
```

Options:

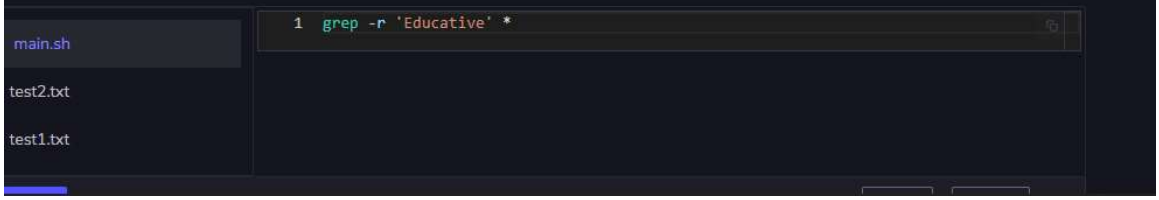
Option	Description
-E (extended regexp)	Causes `grep` to behave like `egrep`.
-F (fixed strings)	Causes `grep` to behave like `fgrep`.
-G (basic regexp)	Causes `grep`, `egrep`, or `fgrep` to behave like the standard **grep** utility.
-r	To search recursively through an entire directory tree (i.e., a directory and all levels of subdirectories within it)
-I	Process a binary file as if it did not contain matching data.
-c	To report the number of times that the pattern has been matched for each file and to not display the actual lines.
-n	To precede each line of output with the number of the line in the text file from which it was obtained.
-v	It matches only those lines that do not contain the given pattern.
-w	To select only those lines that contain an entire

-w	To select only those lines that contain an entire word or phrase that matches the specified pattern.
-x	To select only those lines that match exactly the specified pattern.
-l	To not return the lines containing matches but to only return only the names of the files that contain matches.
-L	It is the opposite of the -l option (and analogous to the -v option) i.e. it will cause grep to return only the names of files that do not contain the specified pattern.

Example:

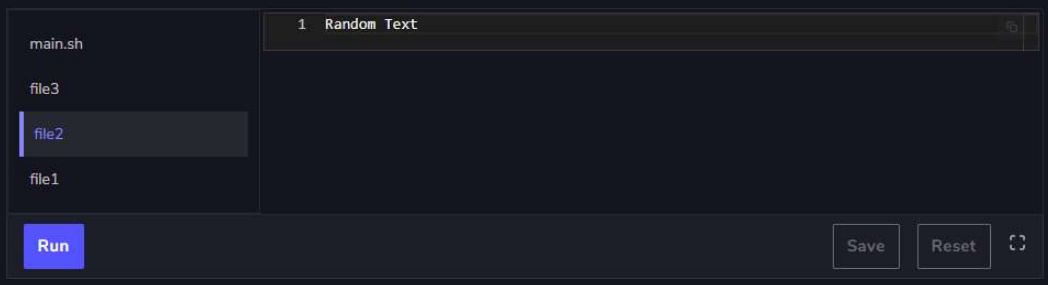
- This would search all files in the current directory and in all of its subdirectories, for every line containing the string “Educative”:

```
rep -r 'Educative' *
```



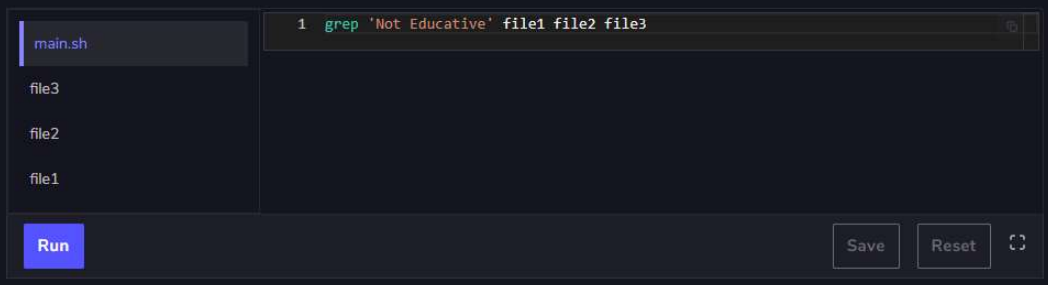
- To search for a word in some respective files:

```
grep Educative file1 file2 file3
```



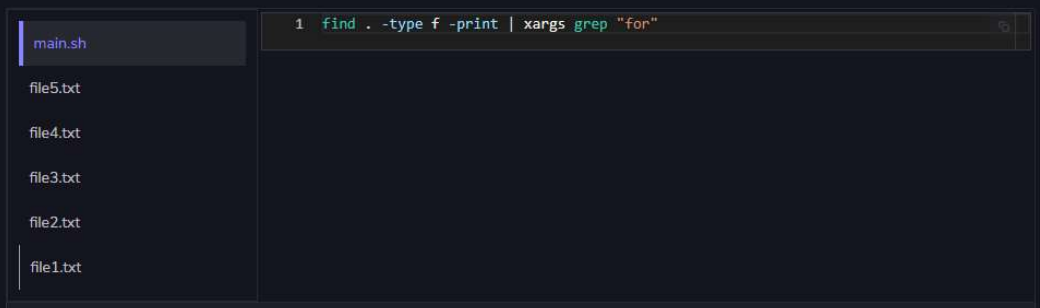
- `grep` can be used to search for a sequence of strings:

```
grep 'Search commands' file1 file2 file3
```



- Using `grep` to find files based on content:

- Using `grep` to find files based on content:#



egrep#

Definition:#

The command `egrep` stands for **"extended global regular expression print"**. It is used for searching particular patterns and is same as `grep` with an `-E` option:

```
grep -E is same as egrep
```

Syntax:#

```
egrep [option(s)] pattern [file(s)]
```

egrep [option(s)] pattern [file(s)]

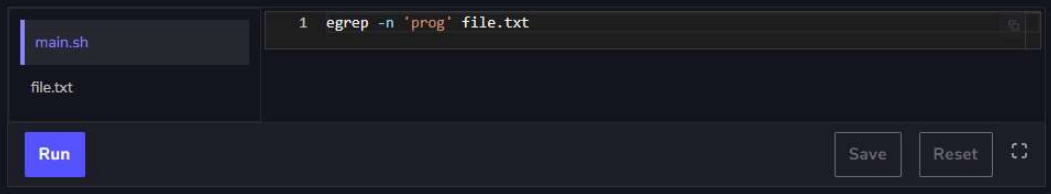
Options:

`egrep` accepts same options as `grep` except `-E` and `-F`.

Examples:

- Identifying every line containing a specific string:

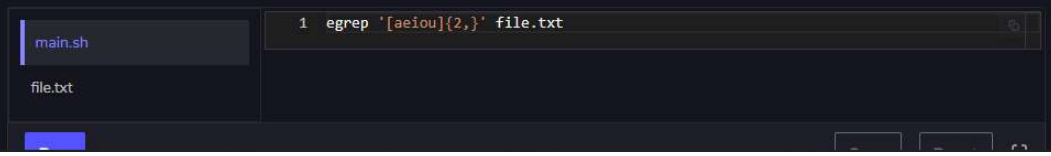
To identify every line containing the string “prog” in **file.txt**, execute the following command. Here, `-n` shows the line numbers along with the results:



Regex Examples

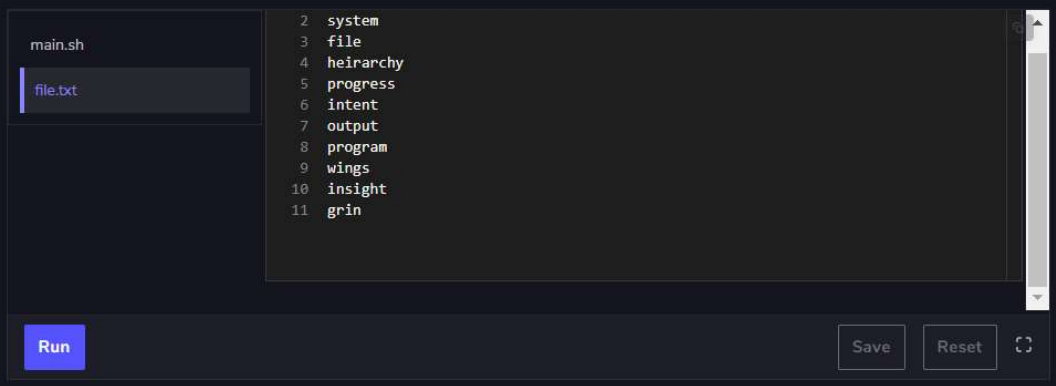
- Finding lines with specific number of vowels:

To find lines with 2 vowels:



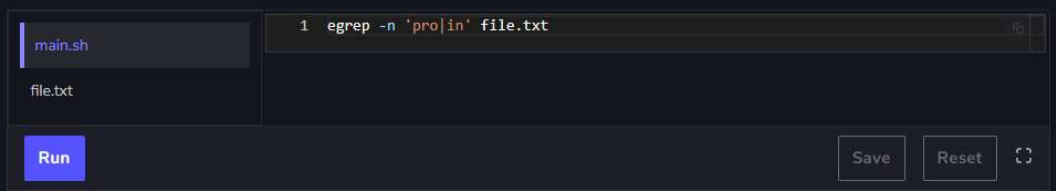
- Finding lines with specific characters in them and those characters don't come at the end of line:

To find the lines that have **'in'** in them and these lines do not end up on **'in'**:



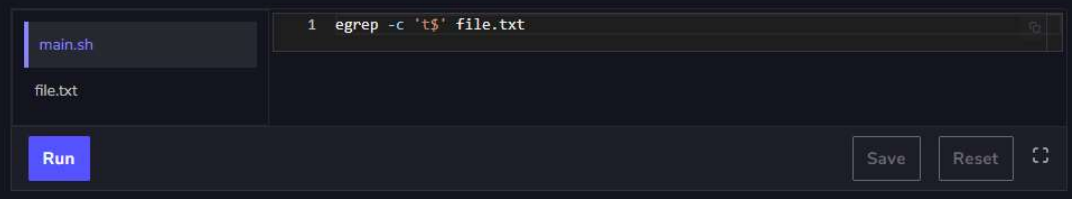
- Finding each line with some sequences of characters:

To find the lines having "pro" or "in":



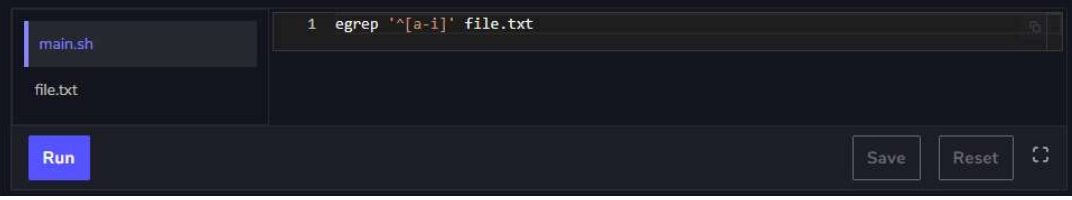
- Finding number of lines with some particular character at the end:

To find the number of lines in **file.txt** with the letter 't' in the end:



- Finding lines beginning with some specific characters:

To find the lines that begin with letters ranging from **'c'** to **'i'**:



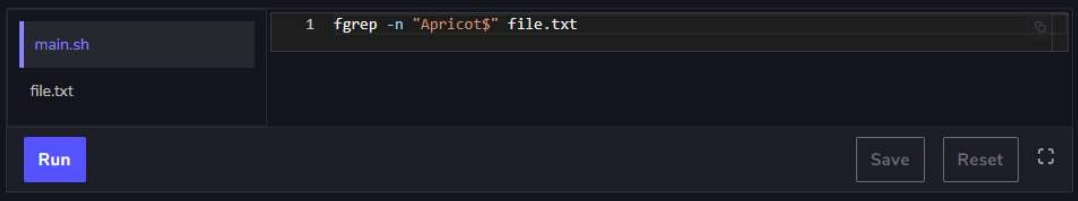
fgrep

Definition:

`fgrep` is used to interpret pattern as a list of **fixed strings** (the whole string is interpreted literally), separated by new lines, Hence, regular expressions can't be used.

Example:

To find the string *"Apricot$"* along with the line number where it's found:



System & Process Commands

Processes

Go through the basic concepts about the system processes and the command to get their status.

> A **process** is a running instance created when a program or a software system is initiated. They can be perceived as programs in action. Processes are often referred as *tasks*.

PID:

Every process has a unique identifier assigned to it at the time of its creation, known as **PID** (*process identifier*).

PPID:

Every process id spawned by its parent process. The **PID** of its parent is called as **PPID**.

ps

Definition:

Short for **"process status"**, the command `ps` gives the status and information about the currently running processes along with their **PID's** (process identification numbers).

Syntax:

```
ps [options]
```

Options:

Option	Description
`-A`	Select all processes. Identical to `-e`.
`-d`	Select all processes except session leaders.
`r`	Restrict the selection to only running processes.
`p pidlist` OR `-p pidlist` OR `--pid pidlist`	Select by process ID.
`--ppid pidlist`	Select by parent process ID.
`-s sesslist` OR `--sid sesslist`	Select by session ID.

Example:

- To see every process running on the system, using the standard syntax:

```
ps -e
```

- To display a process tree:

```
ps -ejh
```

- Print only the name of process ID 21:

```
ps -p 21 -o comm=
```

Jobs: How Do They Differ from Processes

This lesson defines "jobs", how they differ from processes and how to get information about them.

> A **group of processes** running in series or parallel, is considered as a job.

jobs#

Definition:#

The command `jobs` lists the status of all running jobs at some time.

Syntax:#

```
jobs [-lnprs] [jobspec]
jobs -x command [arguments]
```

Options:#

Option	Description
-l	List process IDs in addition to the normal information.
-n	Display information only about jobs that have changed status since the user was last notified of their status.
-p	List only the process ID of the job's process group leader.
-r	Display only running jobs.
-s	Display only stopped jobs.

Example:

- To display all running jobs:

```
jobs
```

- With the -l option, `jobs` displays process IDs in addition to job number:

```
jobs -l
```

Difference between Process and Job

Suppose you have to solve a mathematical situation consisting of 5 different problems, for which you have to launch 5 series of processes in order to solve the whole scenario. Then this task of resolving this whole problem is termed as a job.

> Job is therefore any task performed by the machine where a group of processes perform similar tasks, although the processes may or may not be related.

Getting Root Privileges

This lesson will cover the command to get administrator privileges to do certain tasks that are not allowed for a normal user to perform.

Linux has a robust permission system. It is quite brilliant, in the sense that it offers a clear distinction among ordinary users and a super user. However, to perform certain tasks, especially the ones that affect the files on your system, you might require administrator priveldges. It is recommended to handle root tasks with extreme care in order to prevent any accidental damage to your system.

`sudo` lets you run commands with root access in your own user account.

sudo

Definition:

`sudo` is the short for "superuser do". It allows the user with proper permissions to execute a command as a superuser or another user.

Syntax:

```
sudo -V | -h | -l | -L | -v | -k | -K | -s | [ -H ] [-P ] [-S ] [ -b ] |
     [ -p prompt ] [ -c class|- ] [ -a auth_type ] [-r role ] [-t type ]
     [ -u username|#uid ] command
```

Options:

Option	Meaning
-H	The **-H** (HOME) option sets the HOME environment variable to the home directory of the target user (root by default) as specified in `passwd`. By default, **sudo** does not modify HOME.
-P	The **-P** (preserve group vector) option causes sudo to preserve the current user's group vector unaltered. By default, sudo will initialize the group vector to the list of groups of the target user. The real and effective group IDs, however, are still set to match the target user.
-S	The **-S** (stdin) option causes sudo to read the password from standard input instead of the terminal.
-b	The **-b** (background) option tells sudo to run the given command in the background.
-h	The **-h** (help) option causes sudo to print a usage message and exit.
-l	The **-l** (list) option will print out the commands allowed (and forbidden) the user on the current host.

-v	The **-v** (validate) option will update the user's time-stamp, prompting for the user's password if necessary.
-k	The **-k** (kill) option to sudo invalidates the user's timestamp by setting the time on it to the epoch. The next time sudo is run a password will be required. This option does not require a password and was added to allow a user to revoke sudo permissions from a .logout file.
-s	The **-s** (shell) option runs the shell specified by the `SHELL` environment variable if it is set or the shell as specified in the file `passwd`.

Example:

Get a file listing of an unreadable directory:

```
sudo ls /usr/local/protected
```

To restart the system, execute **shutdown** command as root:

```
sudo shutdown -r now
```

The following command will "kill" sudo authentication for the current user. The next sudo command will require a password.

Example:

Get a file listing of an unreadable directory:

```
sudo ls /usr/local/protected
```

To restart the system, execute **shutdown** command as root:

```
sudo shutdown -r now
```

The following command will "kill" sudo authentication for the current user. The next sudo command will require a password.


```
sudo -k
```

List the contents of `/home/someotheruser/Projs` as the user: **runner**.

```
sudo -u runner ls /home/someotheruser/Projs
```

Killing a Process

Teach yourself how to kill any process in bash in this lesson.

kill#

Definition:#

The `kill` command is used on Linux shell to terminate processes without having to log out or reboot the computer. Hence, it is specifically important to the stability of such systems.

Syntax:#

```
kill [signal or option] PID(s)
```

Example:#

If it is required to terminate the process with **PID** 468:

```
kill 468
```

If this fails, a stronger signal **9** (SIGKILL) should be used.

```
kill 9 468
```

There are more than **60** signals that can be used with the command `kill`. The whole list can be viewed bu using `kill` along with its option `-l`:

```
kill -l
```

Variables and Environment

Variables and their Types

Familiarize yourself with variables in shell environment and their types.

We'll cover the following...

If you have prior programming experience, then you might already be familiar with the basic concepts of Variables. This chapter will briefly summarize all the concepts that you have studied so far and give you hands-on experience on variable assignments. Variables are nothing but placeholders for memory locations. They empower the programmers to quickly store or retrieve data for the successful execution of a program.

In Bash, variables are commonly referred to as **Bash Parameters**. These parameters can be further categorized into three types; *Special Parameters*, *Positional Parameters*, and *Shell Variables*. In this chapter, we will only cover the latter one as it's the most important type of variable. In Bash, it is not necessary to declare the type of a variable so a variable can hold anything that we assign to it be it a number, string, or a character.

Types of Variables

Based on their lifetime, range, and uses, the variables can generally be divided into three categories:

- Shell Variables
- Local Variables
- Environment Variables
- Other Variables

1. Shell Variables#

Shell variables are a combination of local variables which are required by the Shell for proper functioning. Each Shell has its own Shell Variables which are only accessible inside that Shell. These variables are not even local to parent or child Shells.

Examples: `$my_var`

2. Local Variables#

Local variables are set at the command line prompt and are only accessible inside the current shell. They cannot be accessed by child processes/programs which run under the current shell. All user-defined variables are local variables.

Examples: `MY_MESSAGE = HELLO`

3. Environment Variables#

Environment variables, on the other hand, can be accessed anywhere in the Shell and even by the child processes called in a Shell. In Bash, `export` command is used to create Environment variables.

Examples: `$PATH` variable is one of the most important environment variables which specifies a list of directories where programs are stored. It allows installing several versions of the same program and is very convenient to use.

4. Other Variables#

Some other system variables which are set by default by the system are mentioned below:

Variable	Meaning
$USER	Returns the username
$HOSTNAME	Returns the machine name
$$	Returns the Script ID
$0	Returns the name of the Script
$1-9	Returns the first 9 arguments in the Script

Variable Assignment & Expansion

Get yourself accustomed to variable assignment and expansion.

How to set a variable?

Variables usually come in handy when you need to store or retieve something during program execution. To assign a value to a variable we use “=” operator. In Bash, you do not need to declare the type of a variable unlike C++ and JAVA. Which means you can assign any value to a variable whether it is a number, a character or a string of characters.

Some variables are preset by the system but we can set our own variables as well. The syntax to assign value to a variable is given below:

```
name=Jhon
```

Variable Expansion

We use `"$"` sign to give a reference to a variable. This technique is often referred as *Parameter Expansion* in Bash. `$` sign basically tells Bash that there are variables present in the script which needs to be substituted before interpretation of any line so that it can replace the name of that variable with its value.

For example if `my_variable` is a name of the variable, then `$my_variable` is a reference to that variable and returns the value stored in that memory location. `echo` command is used to check the value of any variable. Here’s how we will code this in Bash:

```
name=Jhon
echo "Hello $name, Welcome to Educative!"
```

Run Save Reset

Parameter Expansion Operators#

Sometimes, we might need to modify the value to make the output look more presentable. We use three operator for this purpose:

Operator	Use
`//`	Replace characters, special characters etc.
`%`	Trim the value from end based on any character or delimeter
`#`	Trim the value from start based on any character or delimeter

Important Points#

- It is a good practice to follow variable naming conventions to keep the code simple and readable for other programmers
- Variables are a good way to store information which is reused many times in the program, for example you can store the directory path in a variable and then call it wherever you want instead of writing it again and again
- While assigning a value, there should not be any space between the assignment operator. Bash produces an error if you do so
- Sometimes the `$` sign is not enough, so we add curly brackets with `$` sign to indicate Bash the beginning and ending of variable name, like this: `${name}'s`
- It is a common practice to write Shell variables in lowercase and Environment variables in uppercase format, for example, `$PATH`
- If a variable is uninitialized, then a `null` value is stored in its memory location
- If there's a space between two string, Bash would consider it as two seperate items, so always use Quotations (single or double) in order to avoid this error. So, the correct syntax to initialize a string with spaces would be: `message="Hello John!"`

Terminal is fun (Really)

Train yourself to use command line to your interest in order to install and upgrade different packages. Using GUI for installing applications might be convenient, but contrary, the terminal is a fast approach for this purpose. You will learn how to manage your softwares via apt-get.

What is apt-get?#

> `apt-get` is a friendly command line tool to interact with the packaging system.

It requires administrator privileges (super user accessibility), henceforth, we'll be using `sudo` with `apt-get`.

It can be used to find new packages, install, update and upgrade them.

Using `apt-get` Commands:#

Following are several functions that can be performed via `apt-get`.

Updating Package Database:#

The first task to execute on any Linux system after a fresh install is to run an update on the local packages' database. Without updating, the system wouldn't know of any new available packages.

```
1  sudo apt-get update
```

Upgrading Packages:

Once you have updated all applications, you can now upgrade all of them to their latest versions available.

```
sudo apt-get upgrade
```

To upgrade some specific package:

```
sudo apt-get upgrade <package_name>
```

Difference between `apt-get update` and `apt-get upgrade`:

`apt-get update` only updates the database of packages, i.e. the database will become aware that for some package **abc**, a newer version is available.	`apt-get upgrade` renews the packages to their latest versions.

Installing Packages:

You can install any specific package using `apt-get`. For example, to install the **vim** editor:

```
sudo apt-get install vim
```

Removing Packages:

`apt-get remove` removes binary files of a package and leaves the configuration files untouched.

```
sudo apt-get remove <package_name>
```

`apt get purge` removes anything associated with the package including the conguration files.

```
sudo apt-get purge <package_name>
```

Using `apt-cache` to Search for Packages

If you want to search for packages or get information about them, you can use this command. You don't even have to use `sudo` here.

```
apt-cache search <search_term>
```

Caution: Cool Stuff Ahead!

Enough with the commands already, wanna do something fun? Give this lesson a read to discover what things you can do with just the basic Bash knowledge that you have gained so far.

Uptil now, we only covered the commonly used commands followed by the necessary concepts and terminologies used in Bash. But, the main reason as to why Bash is given so much importance in programmers' world is going to be revealed in these upcoming lessons. Sit tight as you are about to discover the real power of Bash.

Working in Bash

Bash lets you work with a large set of tools and technologies that you can install, update and maintain via terminal and use them to install further packages for third party apps. Some of the most abundantly installed tools are:

- NPM/NVM
- GIT
- Python/R
- Conda
- Docker

In addition to this, we can also download any Shell script online and execute it on our terminal to save a lot of time and effort. As we know, a Shell script is a combination of Bash commands written in order to achieve something. This helps us perform various set of tasks by running just one command to execute that Shell script. There's a huge variety of Shell scripts available online to relief the programmers from the never-ending, hectic procedures of installing the useful softwares and packages required for development. Here are the commands and steps that you need to perform to run a downloaded script:

- Place the script where needed
- Set the execute permissions of file with `chmod` command like this: `chmod a+x my_script.sh`
- Finally, Execute the script like this: `sh my_script.sh`

...and that's it! Interesting, right?

The Real Power of Bash

Get yourself acquainted with the installation of some significant packages.

As mentioned in the previous lesson, Bash could be used with multiple tools to carefully manage packages of any software. In this lesson, we are going to cover multiple technologies and see what Bash is really capable of!

NPM with Bash

NPM is a package manager which is used to install and manage packages for mainly Node.js, a JavaScript framework. It is also used to manage package depencies. It's an online repository that we use to install/update node packages.

1. Install:

To install `npm` via Bash, you need to run this commmand on the terminal. In fact the same format is followed to install anything in Bash:

```
1  sudo apt-get install npm
```

2. How to check if NPM is installed?

You can check the version of your NPM to make sure if it has been installed successfully, just like this:

```
1  npm --version
```

3. Install a package using NPM

To install a package (either local or global), you need to use this command:

```
1  npm install <package_name>
```

Other Commonly Used Package Managers

Using Bash, you can also use different package managers which could be further used to istall, update, search and remove packages and repositories in Linux. Here's a list of examples:

1. Yum

Yum is a package management tool to manage software packages in Linux. For example, we can use `yum` to install `php`, and many other technologies as well, like `nano` editor etc.

```
1  yum -y install php
```

2. Conda#

Conda is another open-source package managers which is used to install, update and remove packages and dependencies for languages like Python, R, Java, ROR etc. It is considered the most convenient way to install softwares like Jupyter and provide environment support for different versions of Python to run successfully on the same machine.

You can install multiple packages at once and even specify their versions in Conda, like this:

```
conda install matplotlib=1.1.1 scipy=1.04.3
```

How to Use Git with Bash

This lesson will brief you about an important and most prevalent software system i.e. Git. Moreover, it'll show you how to install Git on GNU/Linux based systems.

What is Git?#

Git is the most common free and distributed version control system. It aids in non-linear development allowing multiple contributors to work on projects simultaneously.

Git was invented by **Linus Torvalds**, principal developer of Linux kernel.

Installing Git in Linux#

On Ubuntu/Debian:

Most Linux systems used now a days are *Ubuntu/Debian* based. You can install Git on them by entering following commands on the command line:

```
sudo apt-get update
sudo apt-get upgrade
sudo apt-get install git
```

On Fedora:

If you are on *Fedora* or any other close RPM based distribution, you can use `yum` package installer to install **git**:

```
sudo yum install git-core
```

On CentOS:

`yum` can also be used on CentOS:

```
sudo yum install git
```

On Arch Linux:

If you are on Arch Linux, you can use `pacman`, which is another package installer that comes with Arch Linux distributions.

```
sudo pacman -Sy git
```

On Red-Hat Based Linux Systems:

On Red-Hat based linux systems, you can use `yum` to install git:

```
sudo yum upgrade
sudo yum install git
```

Basic Git Commands#

Command	Syntax	Description
git pull	`git pull`	Fetch and merge any commits from the tracking remote branch
git push	`git push [alias] [branch]`	Transmit local branch commits to the remote repository branch.
git add	`git add [filename]`	Add a file to a repository.
git commit	`git commit -m "[descriptive message]"`	Commit all staged objects.
git merge	`git merge [alias]/[branch]`	Merge a remote branch into your current branch to bring it up to date.
git rm	`git rm [file]`	Delete the file from project and stage the removal for commit.
git mv	`git mv [existing-path] [new-path]`	Change an existing file path and stage the move.

An Overview of Commands

Misc. Commands' Cheatsheet

Here's a concise cheatsheet for some miscellaneous commands used in Linux.

Command	Definition	Syntax	Example
bc	`bc` is an arbitrary-precision language for performing math calculations.	`bc [ -hlwsqv ] [long-options] [file]`	To assign the variable `var` a value of **5** and display in on the console: `echo "var=5;var" | bc`
chdir (change directory)	`chdir` is the system command for changing the current working directory.	`chdir direectory_name`	To change directory to "/home/etc/ww": `chdir /home/etc/ww`
df (disk free)	It reports the amount of space used and available on currently mounted filesystems.	`df [option(s)] [device(s)]`	To get information only for the root directory: `df /`
du (disk usage)	It shows the sizes of directories and files.	`du [options] [directories and/or files]`	This will show the sizes of all directories that are in *dir2* that resides in *dir1*: du dir1/dir2

file	This command classifies filesystem objects.	`file [option(s)] object_name(s)`	Information about a file named “file1.txt” can be obtained by: `file file1.txt`
free	It provides information about unused and used memory and swap space.	`free [options]`	To show all of the data in megabytes: `free -m`
gzip	It is used to compress or decompress files.	`gzip [options] [suffix] [filename]`	To decompress a file “file.gz”: `gzip -d file.gz`
halt	This command instructs the hardware to stop all CPU functions.	`halt [option]`	`sudo halt`
hostname	Shows or sets a computer’s host name and domain name.	`hostname [options] [new_host_name]`	To change the hostname to “host2”: `hostname host2`
kdesu	It opens KDE su, the graphical front end for the su command.	`kdesu [-u username] [options] command`	To allow Nautilus (the official file manager for the GNOME desktop) to be run as root during an ordinary user session: `kdesu -c nautilus`

killall	It terminates all processes associated with programs whose names are provided to it as arguments.	`killall [options] program_name(s)`	To abruptly terminate nautilus (which is the official file manager for the GNOME desktop): `killall nautilus`
less	`less` is the more powerful version of `more`. It allows backward movement in the file as well as forward movement. Also, `less` does not have to read the entire input file before starting, so with large input files it starts up faster than text editors like *vi*.	`less <file name>`	`less /var/log/file`
mkfs (make filesystem)	It creates a filesystem on a disk or on a partition thereof.	`mkfs [ -V ] [ -t fstype ] [ fs-options ] filesys [ blocks ]`	This would create an ext2 filesystem on a formatted floppy disk that has been inserted into the first floppy drive: `mkfs /dev/fd0`
more	`more` shows information one page at a time.	`more [filename]`	To view the directory listing of a file, one page at a time: `more file.txt`

mv	It renames and moves files and directories.	`mv [options] argument(s)`	To rename “file1.txt” to “file2.txt”: mv file1.txt file2.txt
pstree	It displays the processes on the system in the form of a tree diagram.	`pstree [options] [pid or username]`	To show only those branches that have been initiated by a user with a username *runner*: `pstree runner`
reboot	It restarts a computer without having to turn the power off and back on.	`reboot [option]`	`reboot`
rsync	It is a fast copying too.	`rsync [options] src [dest]`	`rsync -v /var/lib/rpm/file /root/temp/file`
runlevel	It reports the current and previous runlevels.	`runlevel [utmp]`	It is usually used without any options: `runlevel`
sed (stream editor)	It allows you to filter and transform text.	`sed options [script] [Iimputfile]`	To replace all instances of “cat” with “dog” in ‘file.txt’ `sed 's/cat/dog/g' file.txt`

shred	It destroys files.	`shred [option(s)] file(s)_or_devices(s)`	To securely destroy “file1.txt” and “file2.txt”: `shred file1.txt file2.txt`
shutdown	This command allows you to shutdown a Windows XP, Vista, 7, 8, or 10 computer from the command line, as well as perform additional functions that are not available through Windows.	`shutdown [option]… TIME [MESSAGE]`	`sudo shutdown 5` ; it will shut down system after 5 minutes.
spell	This command checks spellings.	`spell [options] [file_name(s)]`	To spell-chek “file.txt” : `spell file.txt`
strings	It returns each string of printable characters in files.	`strings [options] file_name(s)`	To display all strings in file1.txt that consist of at least 3 characters: `strings -n 3 file1`
su (substitute user)	It changes a login session’s owner without the owner having to first log out of that session.	`su [options] [commands] [-] [username]`	To change the user of current login session to *anna*: `su anna`

tar	This command is used to archive files in tar format. We can also used this command to convert, maintain, and extract tar files.	`tar [option(s)] [archive] [file_name(s)]`	`tar -cf my_files.tar file_1 file_2 file_3`
tr	It translates or deletes characters.	`tr [options] set1 [set2]`	To replace every instance of text-typed *a* with *b*: `tr a b`
unalias	It removes entries from the current user's list of aliases.	`unalias [-a] [alias_name(s)]`	If a user had an alias named p for the `pwd` (i.e., present working directory) command, such alias could be removed with: `unalias p`
uptime	It shows the current time, how long the system has been running since it was booted up, how many user sessions are currently open and the load averages.	`uptime [option]`	`uptime`

vim	It is used for editing any kind of text and is especially suited for editing computer programs.	`vim [options] [filelist]`	To edit a single file: `vim file.txt`
w	`w` shows who is logged into the system and what they are doing.	`w [options] [username1, username2, . . .]`	`w` ; running this command with no arguments shows the list of users along with their ongoing processes.
whatis	It gives very brief descriptions of command line programs and other topics related to Linux and other Unix-like operating systems.	`whatis keyword(s)`	`whatis sudo`
wc (word count)	This command counts the number of lines, words and characters that are contained in text.	`wc [options] [file_name(s)]`	`wc file1 file2`
whereis	It locates the program, source code, and manual page for a command.	`whereis program_name(s)`	`whereis ls`