learnbyexample

Linux Command Line Computing

- 500+ examples
- 200+ exercises

Sundeep Agarwal

Table of contents

Preface

This book aims to teach Linux command line tools and Shell Scripting for beginner to intermediate level users. The main focus is towards managing your files and performing text processing tasks. Topics like system administration and networking won't be discussed.

Prerequisites

You should be familiar with basic computer usage, know fundamental terms like files and directories, how to install programs and so on. You should also be already comfortable with programming basics like variables, loops and functions.

In terms of software, you should have access to the `GNU bash` shell and commonly used Linux command line tools. This could be as part of a Linux distribution or via other means such as a Virtual Machine, WSL (Windows Subsystem for Linux) and so on. More details about the expected working environment will be discussed in the introductory chapters.

You are also expected to get comfortable with reading manuals, searching online, visiting external links provided for further reading, tinkering with the illustrated examples, asking for help when you are stuck and so on. In other words, be proactive and curious instead of just consuming the content passively.

See my curated list on Linux CLI and Shell Scripting for more learning resources.

Conventions

- Code snippets shown are copy pasted from the `bash` shell (version **5.0.17**) and modified for presentation purposes. Some commands are preceded by comments to provide context and explanations, blank lines have been added to improve readability and so on.
- External links are provided throughout the book for you to explore certain topics in more depth.
- The cli-computing repo has all the example files and scripts used in the book. The repo also includes all the exercises as a single file, along with a separate solutions file. If you are not familiar with the `git` command, click the **Code** button on the webpage to get the files.
- See the Setup section for instructions to create a working environment for following along the contents presented in this book.

Acknowledgements

- GNU Manuals — documentation for command line tools and the `bash` shell
- stackoverflow and unix.stackexchange — for getting answers on pertinent questions related to CLI tools
- tex.stackexchange — for help on pandoc and `tex` related questions
- /r/commandline/, /r/linux4noobs/, /r/linuxquestions/ and /r/linux/ — helpful forums
- canva — cover image
- Warning and Info icons by Amada44 under public domain
- carbon — for creating terminal screenshots with highlighted text
- oxipng, pngquant and svgcleaner — optimizing images

Feedback and Errata

I would highly appreciate it if you'd let me know how you felt about this book. It could be anything from a simple thank you, pointing out a typo, mistakes in code snippets, which aspects of the book worked for you (or didn't!) and so on. Reader feedback is essential and especially so for self-published authors.

You can reach me via:

- Issue Manager: https://github.com/learnbyexample/cli-computing/issues
- E-mail: learnbyexample.net@gmail.com
- Twitter: https://twitter.com/learn_byexample

Author info

Sundeep Agarwal is a lazy being who prefers to work just enough to support his modest lifestyle. He accumulated vast wealth working as a Design Engineer at Analog Devices and retired from the corporate world at the ripe age of twenty-eight. Unfortunately, he squandered his savings within a few years and had to scramble trying to earn a living. Against all odds, selling programming ebooks saved his lazy self from having to look for a job again. He can now afford all the fantasy ebooks he wants to read and spends unhealthy amount of time browsing the internet.

When the creative muse strikes, he can be found working on yet another programming ebook (which invariably ends up having at least one example with regular expressions). Researching materials for his ebooks and everyday social media usage drowned his bookmarks, so he maintains curated resource lists for sanity sake. He is thankful for free learning resources and open source tools. His own contributions can be found at https://github.com/learnbyexample.

List of books: https://learnbyexample.github.io/books/

License

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

Code snippets are available under MIT License.

Resources mentioned in the Acknowledgements section above are available under original licenses.

Book version

1.1

See Version_changes.md to track changes across book versions.

Introduction and Setup

Back in 2007, I had a rough beginning as a design engineer at a semiconductor company in terms of utilizing software tools. Linux command line, Vim and Perl were all new to me. In addition to learning about command line tools from colleagues and supervisors, I remember going through and making notes in a photocopied book (unable to recall the title now).

The biggest pain points were not knowing about handy options (for example, `grep --color` to highlight matching portions, `find -exec` to apply commands on filtered files, etc) and tools (for example, `xargs` to workaround limitations of too many command line arguments). And then there were tools like `sed` and `awk` with intimidating syntax. I'm at a loss to reason out why I didn't utilize shell scripts much. I stuck to Perl and Vim instead of learning such handy tools. I also did not know about forums like stackoverflow and unix.stackexchange until after I left my job in 2014.

I started collating what I knew about Linux command line tools when I got chances to conduct scripting course workshops for college students. From 2016 to 2018, I started maintaining my tutorials on Linux command line, Vim and scripting languages as GitHub repos. As you might guess, I then started polishing these materials and published them as ebooks. This is an ongoing process, with **Linux Command Line Computing** being the thirteenth ebook.

This book aims to teach Linux command line tools and Shell Scripting for beginner to intermediate level users. Plenty of examples are provided to make it easier to understand a particular tool and its various features. External links are provided for further reading. Important notes and warnings are formatted to stand out from normal text.

Writing a book always has a few pleasant surprises for me. This time I learned handy options like `mkdir -m` and `chmod =` , got better understanding of many shell features and so on.

This chapter will give a brief introduction to Linux. You'll also see suggestions and instructions for setting up a command line environment to follow along the contents presented in this book.

Linux overview

Quoting selective parts from wikipedia:

> Linux is a family of open-source Unix-like operating systems based on the Linux kernel, an operating system kernel first released on September 17, 1991, by Linus Torvalds. Linux is typically packaged in a Linux distribution.
> Linux was originally developed for personal computers based on the Intel x86 architecture, but has since been ported to more platforms than any other operating system. Because of the dominance of the Linux-based Android on smartphones, Linux also has the largest installed base of all general-purpose operating systems.
> Linux is one of the most prominent examples of free and open-source software collaboration. The source code may be used, modified and distributed commercially or non-commercially by anyone under the terms of its respective licenses, such as the GNU General Public License.

Apart from Linux exposure during my previous job, I've been using Linux since 2014 and it is very well suited for my needs. Compared to Windows, Linux is light weight, secure, stable, fast and more importantly doesn't force you to upgrade hardware. Read the wikipedia article linked above for a more comprehensive coverage about Linux, where it is used and so on.

Linux Distros

Quoting again from wikipedia:

> A Linux distribution (often abbreviated as distro) is an operating system made from a software collection that is based upon the Linux kernel and, often, a package management system. Linux users usually obtain their operating system by downloading one of the Linux distributions, which are available for a wide variety of systems ranging from embedded devices (for example, OpenWrt) and personal computers (for example, Linux Mint) to powerful supercomputers (for example, Rocks Cluster Distribution).

I use Ubuntu, which is beginner friendly. Here are some resources to help you choose a distro:

- /r/linux4noobs wiki — selection guide for noobs
- List of Linux distributions — general information about notable Linux distributions in the form of a categorized list
- DistroWatch — website dedicated to talking about, reviewing and keeping up to date with open source operating systems. This site particularly focuses on Linux distributions and flavours of BSD, though other open source operating systems are sometimes discussed
- Light Weight Linux Distros — uses lower memory and/or has less processor-speed requirements than a more "feature-rich" Linux distribution

Access to Linux environment

You'll usually find installation instructions from the respective distro website you wish to install. Alternatively, you can install Linux on a virtual machine or try it online. Here are some resources to get you started:

- Install Ubuntu desktop
- How to run Ubuntu Desktop on a virtual machine using VirtualBox
- DistroSea — explore and test Linux distributions online

If you are already on Windows or macOS, the following options can be used to get access to Linux tools:

- Git for Windows — provides a Bash emulation used to run Git from the command line
- Windows Subsystem for Linux — compatibility layer for running Linux binary executables natively on Windows
- brew — Package Manager for macOS (or Linux)

 If you are completely new to command line usage, I'd recommend setting up a virtual machine. Or perhaps, a secondary computer that you are free to experiment with. Mistakes in command line can be more destructive compared to the graphical interface. For example, a single space typo can result in data loss, make your machine unusable, etc.

Setup

To follow along the contents presented in this book, you'll need files from my cli-computing repo. Once you have access to a Linux environment, follow the instructions shown below. If the commands used below seem alien to you, wait until you reach the ls section (you'll get a link back to these instructions at that point).

To get the files, you can clone the cli-computing repo using the `git` command or download a `zip` version. You may have to install the `git` command if you don't already have it, for example `sudo apt install git` on Debian-like systems. See https://git-scm.com/downloads for other installation choices.

```
# option 1: use git
$ git clone --depth 1 https://github.com/learnbyexample/cli-computing.git

# option 2: download zip file
# you can also use 'curl -OL' instead of 'wget'
$ wget https://github.com/learnbyexample/cli-computing/archive/refs/heads/master.zip
$ unzip master.zip
$ mv cli-computing-master cli-computing
```

Once you have the files, you'll be able to follow along the commands presented in this book. For example, you'll need to execute the `ls.sh` script for the ls section.

```
$ cd cli-computing/example_files/scripts/
$ ls
cp.sh  file.sh  globs.sh  ls.sh  rm.sh    tar.sh
du.sh  find.sh  grep.sh   mv.sh  stat.sh  touch.sh

$ source ls.sh
$ ls -F
backups/    hello_world.py*  ip.txt     report.log  todos/
errors.log  hi*              projects/  scripts@
```

For sections like the cat command, you'll need to use the sample input files provided in the `text_files` directory.

```
$ cd cli-computing/example_files/text_files/
$ cat greeting.txt
Hi there
Have a nice day
```

Command Line Interface

Command Line Interface (CLI) allows you to interact with the computer using text commands. For example, the `cd` command helps you navigate to a particular directory. The `ls` command shows the contents of a directory. In a graphical environment, you'd use an explorer (file manager) for navigation and directory contents are shown by default. Some tasks can be accomplished in both CLI and GUI environments, while some are suitable and effective only in one of them.

Here are some advantages of using CLI tools over GUI programs:

- automation
- faster execution
- command invocations are repeatable
- easy to save solutions and share with others
- single environment compared to different UI/UX with graphical solutions
- common text interface allows tools to easily communicate with each other

And here are some disadvantages:

- steep learning curve
- syntax can get very complicated
- need to get comfortable with plenty of tools
- typos have a tendency to be more destructive

You can make use of features like command history, shortcuts and autocompletion to help with the plethora of commands and syntax issues. Consistent practice will help to get familiar with the quirks of the command line environment. Commands with destructive potential will usually include options to allow manual confirmation and interactive usage, thus reducing or entirely avoiding the impact of typos.

Chapters

Here's a list of remaining chapters:

Resource lists

This book covers but a tiny portion of Linux command line usage. Topics like system administration and networking aren't discussed at all. Check out the following lists to learn about such topics and discover cool tools:

- Linux curated resources — my collection of resources for Linux command line, shell scripting and other related topics
- Awesome Linux — list of awesome projects and resources that make Linux even more awesome
- Arch wiki: list of applications — sorted by category, helps as a reference for those looking for packages

Command Line Overview

This chapter will help you take the first steps in the command line world. Apart from command examples that you can try out, you'll also learn a few essential things about working in a text environment.

For newbies, the sudden paradigm shift to interacting with the computer using just text commands can be overwhelming, especially for those accustomed to the graphical user interface (GUI). After regular usage, things will start to look systematic and you might realize that GUI is ill suited for repetitive tasks. With continuous use, recalling various commands will become easier. Features like command line history, aliases, tab-completion and shortcuts will help too.

If you've used a scientific calculator, you'd know that it is handy with too many functionalities cramped into a tiny screen and a plethora of multipurpose buttons. Command line environment is something like that, but not limited to just crunching numbers. From managing files to munging data, from image manipulations to working with video, you'll likely find a tool for almost any computing task you can imagine. Always remember that command line tools appeared long before the graphical ones did. The rich history shows its weight in the form of robust tools and the availability of wide variety of applications.

Hello Command Line

Open a Terminal Emulator and type the command as shown below. The `$` followed by a space character at the start is the simple command prompt that I use. It might be different for you. The actual command to type is `echo` followed by a space, then the argument `'Hello Command Line'` and finally press the `Enter` key to execute it. You should get the argument echoed back to you as the command output.

```
$ echo 'Hello Command Line'
Hello Command Line
```

Here's another simple illustration. This time, the command `pwd` is entered by itself (i.e. no arguments). You should get *your* current location as the output. The `/` character separates different parts of the location (more details in the upcoming sections).

```
$ pwd
/home/learnbyexample
```

Next, enter the `exit` command to quit the Terminal session.

```
$ exit
```

If you are completely new to the command line world, try out the above steps a few more times until you feel comfortable with opening a Terminal Emulator, executing commands and quitting the session. More details about the command structure, customizing command prompt, etc will be discussed later.

File System

In Linux, the directory structure starts with the `/` symbol, referred to as the **root** directory. The `man hier` command gives description of the file system hierarchy. Here are some selected examples:

- `/` This is the root directory. This is where the whole tree starts.

- `/bin` This directory contains executable programs which are needed in single user mode and to bring the system up or repair it.
- `/home` On machines with home directories for users, these are usually beneath this directory, directly or not. The structure of this directory depends on local administration decisions (optional).
- `/tmp` This directory contains temporary files which may be deleted with no notice, such as by a regular job or at system boot up.
- `/usr` This directory is usually mounted from a separate partition. It should hold only shareable, read-only data, so that it can be mounted by various machines running Linux.
- `/usr/bin` This is the primary directory for executable programs. Most programs executed by normal users which are not needed for booting or for repairing the system and which are not installed locally should be placed in this directory.
- `/usr/share` This directory contains subdirectories with specific application data, that can be shared among different architectures of the same OS.

Absolute and Relative paths

Quoting wikipedia:

> An **absolute** or **full** path points to the same location in a file system regardless of the current working directory. To do that, it must contain the root directory.

> By contrast, a **relative** path starts from some given working directory, avoiding the need to provide the full absolute path. A filename can be considered as a relative path based at the current working directory. If the working directory is not the file's parent directory, a file not found error will result if the file is addressed by its name.

For example, `/home/learnbyexample` is an absolute path and `../design` is a relative path. You'll learn how paths are used for performing tasks in the coming chapters.

Shells and Terminal Emulators

These terms are often used to interchangeably mean the same thing — a prompt to allow the user to execute commands. However, they are quite different:

- **Shell** is a command line interpreter. Sets the syntax rules for invoking commands, provides operators to connect commands and redirect data, has scripting features like loops, functions and so on
- **Terminal** is a text input/output environment. Responsible for visual details like font size, color, etc

Some of the popular shells are `bash` , `zsh` and `fish` . This book will discuss only the Bash shell. Some of the popular terminal emulators are GNOME Terminal, konsole, xterm and alacritty.

Quoting from wikipedia: Unix shell:

A Unix shell is a command-line interpreter or shell that provides a command line user interface for Unix-like operating systems. The shell is both an interactive command language and a scripting language, and is used by the operating system to control the execution of the system using shell scripts.
Users typically interact with a Unix shell using a terminal emulator; however, direct operation via serial hardware connections or Secure Shell are common for server systems. All Unix shells provide filename wildcarding, piping, here documents, command substitution, variables and control structures for condition-testing and iteration.

Shell features will be discussed in later sections and chapters. For now, open a terminal and try out the following commands:

```
$ cat /etc/shells
# /etc/shells: valid login shells
/bin/sh
/bin/dash
/bin/bash
/bin/rbash

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

In the above example, the `cat` command is used to display the contents of a file and the `echo` command is used to display the contents of a variable. `SHELL` is an environment variable containing the full path to the shell.


The output of the above commands might be different for you. And as mentioned earlier, your command prompt might be different than `$` . For now, you can ignore it. Or, you could type `PS1='$ '` and press the `Enter` key to set the prompt for the current session.

Further Reading

- unix.stackexchange: What is the exact difference between a 'terminal', a 'shell', a 'tty' and a 'console'?
- wikipedia: Comparison of command shells
- unix.stackexchange: Difference between login shell and non-login shell
- Features and differences between various shells
- Syntax comparison on different shells with examples
- Shell, choosing shell and changing default shells

Unix Philosophy

Quoting from wikipedia: Unix Philosophy:

- Write programs that do one thing and do it well.
- Write programs to work together.
- Write programs to handle text streams, because that is a universal interface.

These principles do not strictly apply to all the command line tools, but it is good to be aware

of them. As you get familiar with working from the command line, you'll be able to appreciate these guidelines better.

Command Structure

It is not necessary to fully understand the commands used in this chapter, just the broad strokes. The examples are intended to help you get a feel for the basics of using command options and arguments.

Command invocation without any options or arguments:

- `clear` clear the terminal screen
- `date` show the current date and time

Command with options (flags):

- `ls -l` list directory contents in a long listing format
- `ls -la` list directory contents including hidden files in long listing format
  - two short options `-l` and `-a` are combined together here as `-la`
- `df -h` report file system disk space usage sizes in human readable format
- `df --human-readable` same as `df -h` but using long option

Command with arguments:

- `mkdir project` create a directory named `project` in the current working directory
- `man sort` manual page for the `sort` command
- `diff file1.txt file2.txt` display differences between the two input files
- `wget https://s.ntnu.no/bashguide.pdf` download a file from the internet
  - the link passed to `wget` in the above example is real, visit BashGuide for details

Command with both options and arguments:

- `rm -r project` remove (delete) the `project` directory recursively
- `paste -sd, ip.txt` serialize all lines from the input file to a single line using `,` as the delimiter

Single quotes vs Double quotes:

- **Single quotes** preserves the literal value of each character within the quotes
- **Double quotes** preserves the literal value of all characters within the quotes, with the exception of `$`, `` ` ``, `\`, and, when history expansion is enabled, `!`

```
# no character is special within single quotes
$ echo '$SHELL'
$SHELL

# $ is special within double quotes, used to interpolate a variable here
$ echo "Full path to the shell: $SHELL"
Full path to the shell: /bin/bash
```

More details and other types of quoting will be discussed in the Shell Features chapter.

Command Network

One of the *Unix Philosophy* seen earlier mentioned commands working together. The shell provides several ways to do so. A commonly used feature is redirecting the output of a command —

as input of another command, to be saved in a file and so on.

- to another command
  - `du -sh * | sort -h` calculate size of files and folders in human-readable format using `du` and then sort them using a tool specialized for that task
- to a file
  - `grep 'pass' *.log > pass_list.txt` write the results to a file instead of displaying on the terminal (if the file already exists, it gets overwritten)
  - `grep 'error' *.log >> errors.txt` append the results to the given file (creates a new file if necessary)
- to a variable
  - `d=$(date)` save command output in a variable named `d`

Many more of such shell features will be discussed in later chapters.

Scripting

Not all operations can be completed using a one-liner from the terminal. In such cases, you can save the instructions in a text file and then execute them. Open your favorite text editor and write the three lines shown below:

```
$ cat cmds.sh
echo 'hello world'
echo 'how are you?'
seq 3
```

As an alternate to using a text editor, you can use either of the commands shown below to create this file.

```
# assuming 'echo' supports '-e' option in your environment
$ echo -e "echo 'hello world'\necho 'how are you?'\nseq 3" > cmds.sh

# a more portable solution using the builtin 'printf' command
$ printf "echo 'hello world'\necho 'how are you?'\nseq 3\n" > cmds.sh
```

The script file is named `cmds.sh` and has three commands in three separate lines. One way to execute the contents of this file is by using the `source` command:

```
$ source cmds.sh
hello world
how are you?
1
2
3
```


Your Linux distro is likely to have an easy to use graphical text editor such as the `GNOME Text Editor` and `mousepad`. See wiki.archlinux: text editors for a huge list of editors to choose from.

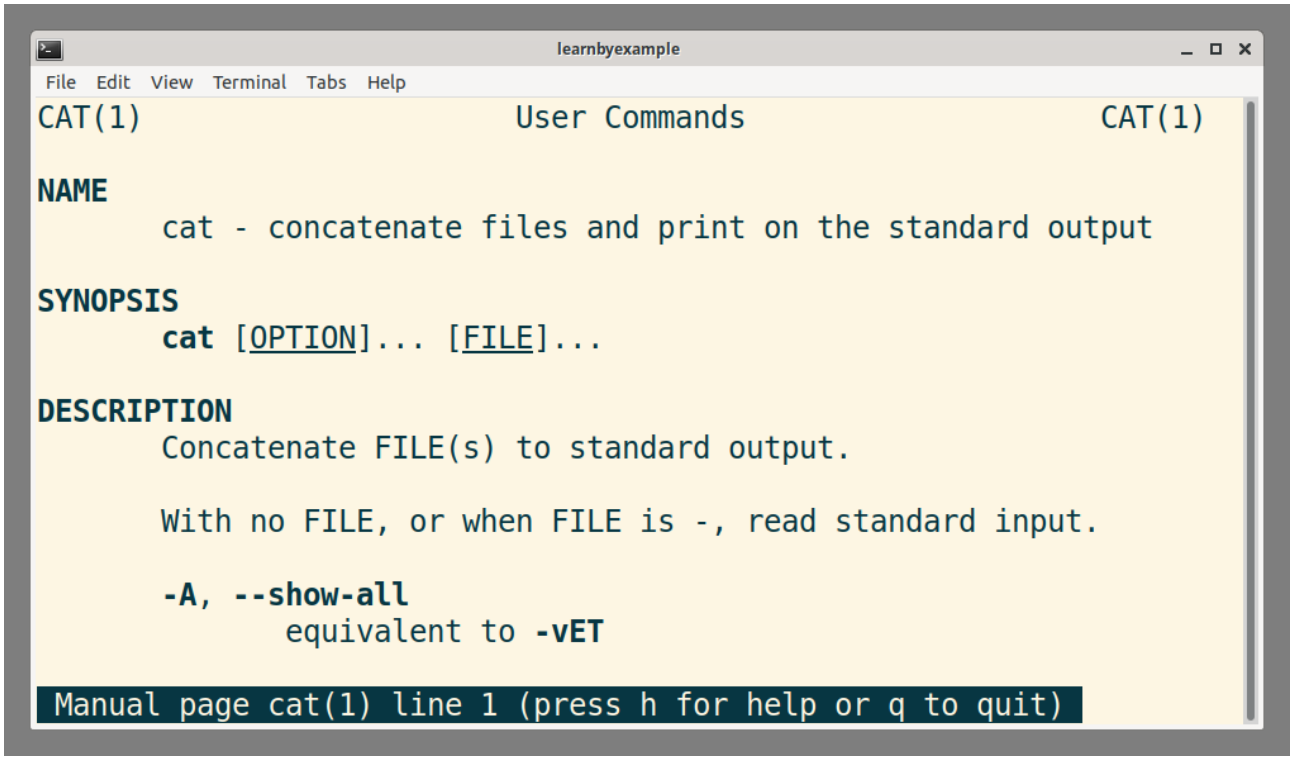

The Shell Scripting chapter will discuss scripting in more detail.

Command Help

Most distros for personal use come with documentation for commands already installed. Learning how to use manuals from the terminal is handy and there are ways to get specific information as well.

man

The `man` command is an interface to view manuals from within the terminal itself. This uses a `pager` (which is usually the `less` command) to display the contents. You could categorize these commands as terminal user interface (TUI) applications. As an example, type `man cat` and you should see something like the screenshot shown below:



Since the documentation has several lines that doesn't completely fit within the terminal window, you will get only the starting part of the manual. You have several options to navigate:

- `↑` and `↓` arrow keys to move up and down by a line
  - you can also use `k` and `j` keys (same keys as those used by the Vim text editor)
- `f` and `b` keys to move forward and backward by a screenful of content
  - `Space` key also moves forward by a screen
- mouse scroll moves up and down by a few lines
- `g` or `Home` go to the start of the manual
- `G` or `End` go to the end of the manual
- `/pattern` followed by `Enter` search for the given pattern in the forward direction
- `?pattern` followed by `Enter` search for the given pattern in the backward direction
- `n` go to the next match
- `N` go to the previous match
- `q` quit

As you might have noticed in the screenshot above, you can use `h` for help about the `less` command itself. Here are some useful tips related to documentation:

- `man man` gives information about the `man` command itself
- `man bash` will give you the manual page for the `bash` shell
  - since this is very long, I'd recommend using the online GNU Bash manual
- `man find | gvim -` open the manual page in your favorite text editor
- `man -k printf` search the short descriptions in all of the manual pages for the string `printf`
  - you can also use the `apropos` command instead of `man -k`
- `wc --help` many commands support the `--help` option to give succinct details like options and syntax
  - also, these details will be displayed on the terminal itself, no need to deal with the `pager` interface


See also unix.stackexchange: How do I use man pages to learn how to use commands? and unix.stackexchange: colors in man pages.


The Linux manual pages are usually shortened version of the full documentation. You can use the `info` command to view the complete documentation for GNU tools. `info` is also a TUI application, but with different key configuration compared to the `man` command. See GNU Manuals Online if you'd prefer to read them from a web browser. You can also download them in formats like PDF for offline usage.

type

For certain operations, the shell provides its own set of commands, known as builtin commands. The `type` command displays information about a command like its path, whether it is a builtin, alias, function and so on.

```
$ type cd
cd is a shell builtin
$ type sed
sed is /bin/sed
$ type type
type is a shell builtin

# multiple commands can be given as arguments
$ type pwd awk
pwd is a shell builtin
awk is /usr/bin/awk
```

As will be discussed in the Shell Customization chapter, you can create aliases to customize command invocations. You can use the `type` command to reveal the nature of such aliases. Here are some examples based on aliases I use:

```
$ type p
p is aliased to 'pwd'

$ type ls
ls is aliased to 'ls --color=auto'
```

The `type` command formats the command output with a backtick at the start and a single quotes at the end. That doesn't play well with syntax highlighting, so I've changed the backtick to single quotes in the above illustration.


See also unix.stackexchange: What is the difference between a builtin command and one that is not?

help

The `help` command provides documentation for builtin commands. Unlike the `man` command, the entire text is displayed as the command output. A help page in the default format is shown below. You can add `-m` option if you want the help content in a pseudo-manpage format.

```
$ help pwd
pwd: pwd [-LP]
    Print the name of the current working directory.

    Options:
      -L        print the value of $PWD if it names the current working directory
      -P        print the physical directory, without any symbolic links

    By default, 'pwd' behaves as if '-L' were specified.

    Exit Status:
    Returns 0 unless an invalid option is given or the current directory
    cannot be read.
```

You can use the `-d` option to get a short description of the command:

```
$ help -d compgen
compgen - Display possible completions depending on the options.
```


Use `help help` for documentation on the `help` command. If you use `help` without any argument, it will display all the internally defined shell commands.

whatis and whereis

Here are some more ways to get specific information about commands:

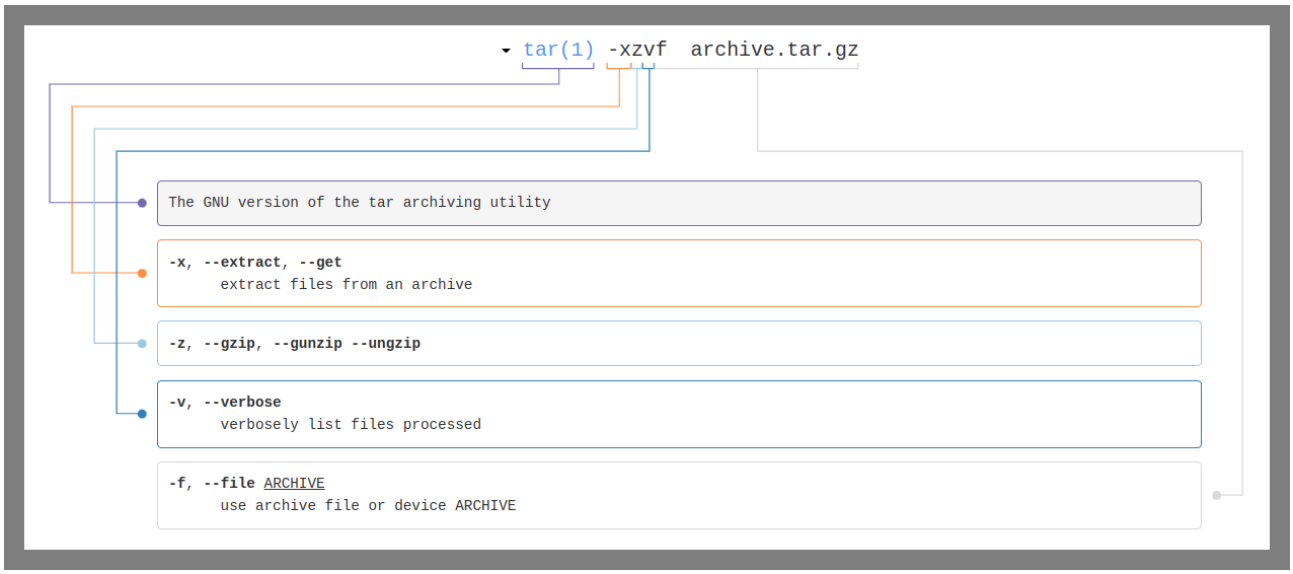
- `whatis` displays one-line manual page descriptions
- `whereis` locates the binary, source, and manual page files for a command

```
$ whatis grep
grep (1)             - print lines that match patterns

$ whereis awk
awk: /usr/bin/awk /usr/lib/x86_64-linux-gnu/awk /usr/share/awk
/usr/share/man/man1/awk.1.gz
```

ch

explainshell is a web app that shows the help text that matches each argument of the command you type in the app. For example, a screenshot for tar -xzvf archive.tar.gz is shown below:



Inspired by this app, I wrote a Bash script ch to extract information from `man` and `help` pages. Here are some examples:

```
$ ch ls -vX
       ls - list directory contents

       -v     natural sort of (version) numbers within text

       -X     sort alphabetically by entry extension

$ ch type -a
    type - Display information about command type.

      -a        display all locations containing an executable named NAME;
                includes aliases, builtins, and functions, if and only if
                the '-p' option is not also used
```

Further Reading

- Linux man pages — one of several websites that host man pages online
- ArchWiki — comprehensive documentation for Arch Linux and other distributions
- Debian Reference — broad overview of the Debian system, covers many aspects of system administration through shell-command examples

Shortcuts and Autocompletion

There are several shortcuts you can use to be productive at the command line. These will be discussed in the Shell Customization chapter. Here are some examples to give an idea:

- `Ctrl+u` delete everything to the left of the cursor
- `Ctrl+k` delete from the current character to the end of the line
- `Ctrl+c` abort the currently typed command

- `Ctrl+l` clear the terminal screen and move the prompt to the top, any characters typed as part of the current command will be retained
- `↑` and `↓` arrow keys to navigate previously used commands from the history
  - `Ctrl+p` and `Ctrl+n` can also be used instead of arrow keys
  - you can modify the command before executing such lines from the history

The tab key helps you autocomplete commands, aliases, filenames and so on, depending on the context. If there is only one possible completion, it will be done on single tab press. Otherwise, you can press the tab key twice to get a list of possible matches (if there are any). Here's an example of completing a file path with multiple tab key presses at various stages. Not only does it saves time, it also helps to avoid typos since you are simultaneously verifying the path.

```
# pressing tab after typing '/e' will autocomplete to '/etc/'
$ ls /etc/

# pressing tab after 'di' will autocomplete to 'dict'
$ ls /etc/dict
# pressing tab twice will show all possible completions
$ ls /etc/dict
dictd/               dictionaries-common/

# type 'i' and press tab to get 'dictionaries-common'
$ ls /etc/dictionaries-common/

# type 'w' and press tab to get 'words'
$ ls /etc/dictionaries-common/words
```

The character at which the tab key is pressed in the above example has been cherry picked for illustration purposes. The number of steps would increase if you try pressing tab after each character. With experience, using the tab key for autocompletion will become a natural part of your command line usage.


You can set an option to combine the features of single and double tab presses into a single tab press. This will be discussed in the Shell Customization chapter.

Real world use cases

If the command line environment only had file managing features, I'd still use it. Given the wide variety of applications available, I can't imagine going back to using a different GUI application for each use case. My primary work is writing ebooks, blog posts and recording videos. Here are the major CLI tools I use:

- text processing using `head`, `tail`, `sort`, `grep`, `sed`, `awk` and so on (you'll learn about these commands in later chapters)
- git — version control
- pandoc — generating PDF/EPUB book versions from markdown files
- mdBook — web version of the books from markdown files
- zola — static site generator
- ImageMagick — image processing like resizing, adding borders, etc
- oxipng, pngquant and svgcleaner — optimizing images
- auto-editor — removing silent portions from video recordings

- FFmpeg — video processing, padding for example (`FFmpeg` is also a major part of the `auto-editor` solution)

Some of these workflows require additional management, for which I write shell functions or scripts. I do need GUI tools as well, for example, web browser, image viewer, PDF/EPUB viewers, SimpleScreenRecorder and so on. Some of these can be handled from within the terminal too, but I prefer GUI for such cases. I do launch some of them from the terminal, primarily for providing the file or url to be opened.

You might wonder what advantage does the command line provide for processing images and videos? Apart from being faster, the custom parameters (like border color, border size, quality percentage, etc) are automatically saved as part of the scripts I create. After that, I can just use a single call to the script instead of waiting for a GUI application to open, navigating to the required files, applying custom parameters, saving them after all the required processing is done, closing the application, etc. Also, that single script can use as many tools as needed, whereas with GUI you'll have to repeat such steps with different applications.

Exercises

All the exercises are also collated together in one place at exercises.md. For solutions, see exercise-solutions.md.

1) By default, is `echo` a shell builtin or external command on your system? What command could you use to get an answer for this question?

2) What output do you get for the command shown below? Does the documentation help understand the result?

```
$ echo apple     42 'banana     100'
```

3) Go through bash manual: Tilde Expansion. Is `~/projects` a relative or an absolute path? See this unix.stackexchange thread for answers.

4) Which key would you use to get help while the `less` command is active?

5) How would you bring the 50th line to the top of the screen while viewing a `man` page (assume `less` command is the `pager`)?

6) What does the `Ctrl+k` shortcut do?

7) Briefly explain the role of the following shell operators:

a) `|`
b) `>`
c) `>>`

8) The `whatis` command displays one-line descriptions about commands. But it doesn't seem to work for `whatis type` . What should you use instead?

```
$ whatis cat
cat (1)              - concatenate files and print on the standard output

$ whatis type
type: nothing appropriate.
```

```
# ???
type - Display information about command type.
```

9) What is the role of the `/tmp` directory?

10) Give an example each for absolute and relative paths.

11) When would you use the `man -k` command?

12) Are there differences between the `man` and `info` pages?

Managing Files and Directories

This chapter presents commands to do things that are typically handled by a file manager in GUI (also known as file explorer). For example, viewing contents of a directory, navigating to other directories, cut/copy/paste files, renaming files and so on. Some of the commands used for these purposes are provided by the shell itself.

As a good practice, make it a habit to go through the documentation of the commands you encounter. Getting used to looking up documentation from the command line will come in handy whenever you are stuck. You can also learn and experiment with options you haven't used yet.


The example_files directory has the scripts used in this chapter. See the Setup section for instructions to create the working environment.

Builtin and External commands

From bash manual: What is a shell?

> Shells also provide a small set of built-in commands (builtins) implementing functionality impossible or inconvenient to obtain via separate utilities. For example, `cd`, `break`, `continue`, and `exec` cannot be implemented outside of the shell because they directly manipulate the shell itself. The `history`, `getopts`, `kill`, or `pwd` builtins, among others, could be implemented in separate utilities, but they are more convenient to use as builtin commands.

Many of the commands needed for everyday use are external commands, i.e. not part of the shell. Some builtins, `pwd` for example, might also be available as external command on your system (and these might have differences in features too). In such cases the builtin version will be executed by default, which you can override by using the path of the external version.

You can use the `type` command to check if the tool you are using is a builtin or an external command. The `type` command is a shell builtin, and provides other features too (which will be discussed later). You can use the `-a` option to get *all* details about the given command.

```
$ type -a cd
cd is a shell builtin

$ type -a ls
ls is /bin/ls

$ type -a pwd
pwd is a shell builtin
pwd is /bin/pwd
```


To look up documentation, use the `help` command for builtins and `man` for external commands (or `info` for complete documentation, where applicable). Use `help help` and `man man` for their own documentation.

Typing just `help` will give the list of builtins, along with the command's syntax.

pwd

`pwd` is a shell builtin command to get the current working directory. This helps to orient yourself with respect to the filesystem. The absolute path printed is often handy to copy-paste elsewhere, in a script for example. Some users prefer their terminal emulators and/or shell prompt to always display the current working directory.

```
$ pwd
/home/learnbyexample
```

cd

`cd` is another shell builtin. This helps to change the current working directory. Here's an example of changing the current working directory using an absolute path:

```
$ pwd
/home/learnbyexample

# specifying / at end of the path is optional
$ cd /etc
$ pwd
/etc
```

You can use `-` as an argument to go back to the previous working directory. Continuing from the previous example:

```
$ cd -
/home/learnbyexample
```

Most commands will treat strings starting with `-` as a command option. You can use `--` to tell commands that all the following arguments should *not* be treated as options even if they start with `-`. For example, if you have a folder named `-oops` in the current working directory, you can use `cd -- -oops` to switch to that directory.

Relative paths are well, relative to the current working directory:

- `.` refers to the current directory
- `..` refers to the directory one hierarchy above (i.e. the parent directory)
- `../..` refers to the directory two hierarchies above and so on
- `cd ./-` will help you to switch to a directory named `-` in the current location
    - you cannot use `cd -` since that'll take you to the previous working directory

```
$ pwd
/home/learnbyexample

# go one hierarchy above
$ cd ..
$ pwd
```

```
/home

# change to 'learnbyexample' present in the current directory
# './' is optional in this case
$ cd ./learnbyexample
$ pwd
/home/learnbyexample

# go two hierarchies above
$ cd ../..
$ pwd
/
```

You can switch to the home directory using `cd` or `cd ~` or `cd ~/` from anywhere in the filesystem. This is determined by the value stored in the `HOME` shell variable. See also bash manual: Tilde Expansion.

```
$ pwd
/
$ echo "$HOME"
/home/learnbyexample

$ cd
$ pwd
/home/learnbyexample
```

clear

You can use this command to clear the terminal screen. By default, the `clear` command will move the prompt to the top of the terminal as well as try to remove the contents of the scrollback buffer. You can use the `-x` option if you want to retain the scrollback buffer contents.

The `Ctrl+l` shortcut will also move the prompt line to the top of the terminal. It will retain any text you've typed on the prompt line and scrollback buffer contents won't be cleared.

ls

When you use a file explorer GUI application, you'll automatically see the directory contents. And such GUI apps typically have features to show file size, differentiate between files and folders and so on. `ls` is the equivalent command line tool with a plethora of options and functionality related to viewing the contents of directories.

As mentioned earlier, the example_files directory has the scripts used in this chapter. You can source the `ls.sh` script to follow along the examples shown in this section. See the Setup section if you haven't yet created the working environment.

```
# first, cd into the 'scripts' directory
$ cd cli-computing/example_files/scripts

$ ls
cp.sh  file.sh  globs.sh  ls.sh  rm.sh    tar.sh
du.sh  find.sh  grep.sh   mv.sh  stat.sh  touch.sh

# 'ls.sh' script will create a directory named 'ls_examples'
# and automatically change to that directory as well
$ source ls.sh
$ pwd
/home/learnbyexample/cli-computing/example_files/scripts/ls_examples
```

By default, the current directory contents are displayed. You can pass one or more paths as arguments. Here are some examples:

```
$ ls
backups     hello_world.py  ip.txt    report.log  todos
errors.log  hi              projects  scripts

# example with a single path argument
$ ls /sys
block  class  devices   fs          kernel  power
bus    dev    firmware  hypervisor  module

# multiple paths example
# directory listings will be preceded by their names
$ ls projects backups ip.txt
ip.txt

backups:
bookmarks.html  dot_files

projects:
calculator  tictactoe
```

You can use the `-1` option (`1` as in numeric one, not the letter `l` which does something else) to list the contents in a single column:

```
$ ls -1 backups
bookmarks.html
dot_files
```

The `-F` option appends a character to each filename indicating the file type (if it is other than a regular file):

- `/` directory

- `*` executable file
- `@` symbolic link
- `|` FIFO
- `=` socket
- `>` door

```
$ ls -F
backups/    hello_world.py*  ip.txt     report.log  todos/
errors.log  hi*              projects/  scripts@
```

If you just need to distinguish between files and directories, you can use the `-p` option:

```
$ ls -p
backups/    hello_world.py  ip.txt     report.log  todos/
errors.log  hi              projects/  scripts
```

You can also use the `--color` option to visually distinguish file types:

The `-l` option displays the contents using a long listing format. You'll get details like file permissions, ownership, size, timestamp and so on. The first character of the first column distinguishes file types as `d` for directories, `-` for regular files, `l` for symbolic links, etc. Under each directory listing, the first line will display the total size of the entries (in terms of KB).

```
$ ls -l hi
-rwxrwxr-x 1 learnbyexample learnbyexample 21 Dec  5  2019 hi

# you can add -G option to avoid the group column
$ ls -lG
total 7516
drwxrwxr-x 3 learnbyexample    4096 Feb  4 09:23 backups
-rw-rw-r-- 1 learnbyexample   12345 Jan  1 03:30 errors.log
-rwxrwxr-x 1 learnbyexample      42 Feb 29  2020 hello_world.py
-rwxrwxr-x 1 learnbyexample      21 Dec  5  2019 hi
-rw-rw-r-- 1 learnbyexample      10 Jul 21  2017 ip.txt
drwxrwxr-x 4 learnbyexample    4096 Mar  5 11:21 projects
-rw-rw-r-- 1 learnbyexample 7654321 Jan  1 01:01 report.log
lrwxrwxrwx 1 learnbyexample      13 May  7 15:17 scripts -> ../../scripts
drwxrwxr-x 2 learnbyexample    4096 Apr  6 13:19 todos
```

Note that the timestamps showing hours and minutes instead of year depends on the relative difference with respect to the current time. So, for example, you might get `Feb 4 2022` instead of `Feb 4 09:23` .

Use the `-h` option to show file sizes in human readable format (default is byte count).

```
$ ls -lG report.log
-rw-rw-r-- 1 learnbyexample 7654321 Jan  1 01:01 report.log

$ ls -lhG report.log
-rw-rw-r-- 1 learnbyexample 7.3M Jan  1 01:01 report.log
```

You can use the `-s` option instead of long listing if you only need allocated file sizes and names:

```
$ ls -1sh errors.log report.log
 16K errors.log
7.4M report.log
```

There are several options for changing the order of listing:

- `-t` sorts by timestamp
- `-S` sorts by file size (not suitable for directories)
- `-v` version sorting (suitable for filenames with numbers in them)
- `-X` sorts by file extension (i.e. characters after the last `.` in the filename)
- `-r` reverse the listing order

```
$ ls -lGhtr
total 7.4M
-rw-rw-r-- 1 learnbyexample   10 Jul 21  2017 ip.txt
-rwxrwxr-x 1 learnbyexample   21 Dec  5  2019 hi
-rwxrwxr-x 1 learnbyexample   42 Feb 29  2020 hello_world.py
-rw-rw-r-- 1 learnbyexample 7.3M Jan  1 01:01 report.log
-rw-rw-r-- 1 learnbyexample  13K Jan  1 03:30 errors.log
drwxrwxr-x 3 learnbyexample 4.0K Feb  4 09:23 backups
drwxrwxr-x 4 learnbyexample 4.0K Mar  5 11:21 projects
drwxrwxr-x 2 learnbyexample 4.0K Apr  6 13:19 todos
lrwxrwxrwx 1 learnbyexample   13 May  7 15:17 scripts -> ../../scripts
```

Filenames starting with `.` are considered as hidden files and these are NOT shown by default. You can use the `-a` option to view them. The `-A` option is similar, but doesn't show the special `.` and `..` entries.

```
# . and .. point to the current and parent directories respectively
$ ls -aF backups/dot_files/
./  ../  .bashrc  .inputrc  .vimrc

# -A will exclude the . and .. entries
$ ls -A backups/dot_files/
.bashrc  .inputrc  .vimrc
```

The `-R` option recursively lists sub-directories as well:

```
$ ls -ARF
.:
backups/    hello_world.py*  .hidden  projects/   scripts@
errors.log  hi*              ip.txt   report.log  todos/

./backups:
bookmarks.html  dot_files/

./backups/dot_files:
.bashrc  .inputrc  .vimrc

./projects:
calculator/  tictactoe/

./projects/calculator:
calc.sh

./projects/tictactoe:
game.py

./todos:
books.txt  outing.txt
```

Often you'd want to list only specific files or directories based on some criteria, file extension for example. The shell provides a matching technique called **globs** or **wildcards**. Some simple examples are shown below (see the wildcards section for more details).

`*` is a placeholder for zero or more characters:

```
# *.py *.log will give filenames ending with '.py' or '.log'
$ echo *.py *.log
hello_world.py errors.log report.log

# glob expansion can be prevented by using quotes
$ echo '*.py' *.log
*.py errors.log report.log

# long list only files ending with '.log'
$ ls -lG *.log
-rw-rw-r-- 1 learnbyexample   12345 Jan  1 03:30 errors.log
-rw-rw-r-- 1 learnbyexample 7654321 Jan  1 01:01 report.log
```

`[]` helps you specify a set of characters to be matched once. For example, `[ad]` matches `a` or `d` once. `[c-i]` matches a range of characters from `c` to `i` .

```
# entries starting with 'c' to 'i'
$ echo [c-i]*
errors.log hello_world.py hi ip.txt

$ ls -1sh [c-i]*
 16K errors.log
4.0K hello_world.py
```

```
4.0K hi
4.0K ip.txt
```

 As shown in the above examples, globs are expanded by the shell. Beginners often associate globs as something specific to the `ls` command, which is why I've deliberately used `echo` as well in the above examples.

You can use the `-d` option to *not* show directory contents:

```
$ echo b*
backups
# since backups is a directory, ls will show its contents
$ ls b*
bookmarks.html  dot_files
# -d will show the directory entry instead of its contents
$ ls -d b*
backups

# a handy way to get only the directory entries
$ echo */
backups/ projects/ scripts/ todos/
$ ls -1d */
backups/
projects/
scripts/
todos/
```

 I hope you have been judiciously taking notes, since there are just too many commands and features. For example, note down all the options discussed in this section. And then explore the output from the `ls --help` command.

Further Reading

- mywiki.wooledge: avoid parsing output of ls
- unix.stackexchange: why not parse ls?
- unix.stackexchange: What are ./ and ../ directories?

tree

The `tree` command displays the contents of a directory recursively, in a hierarchical manner. Here's a screenshot of using `tree -a` from the `ls_examples` sample directory seen in the previous section. The `-a` option is used to show the hidden files as well.


learnbyexample
File Edit View Terminal Tabs Help
$ tree -a
.
├── backups
│   ├── bookmarks.html
│   └── dot_files
│   ├── .bashrc
│   ├── .inputrc
│   └── .vimrc
├── errors.log
├── hello_world.py
├── hi
├── .hidden
├── ip.txt
├── projects
│   ├── calculator
│   │   └── calc.sh
│   └── tictactoe
│   └── game.py
├── report.log
├── scripts -> ../../scripts
└── todos
    ├── books.txt
    └── outing.txt
7 directories, 14 files
$


You might have to install this command. `sudo apt install tree` can be used to get this command on Debian-like distributions.

mkdir

The `mkdir` command helps you to create new directories. You can pass one or more paths along with the name of the directories you want to create. Quote the names if it can contain shell special characters like space, `*` and so on.


Create a practice directory for this section:

```
$ mkdir practice_mkdir
$ cd practice_mkdir
```

Here's an example of creating multiple directories:

```
$ mkdir reports 'low power adders'

$ ls -1
'low power adders'
reports
```

The `-p` option will help you to create multiple directory hierarchies in one shot:

```
# error because 'a' and 'a/b' paths do not exist yet
$ mkdir a/b/c
mkdir: cannot create directory ‘a/b/c’: No such file or directory

# -p is handy in such cases
$ mkdir -p a/b/c

$ tree
.
├── a
│   └── b
│       └── c
├── low power adders
└── reports

5 directories, 0 files
```

The `-p` option has another functionality too. It will not complain if the directory you are trying to create already exists. This is especially helpful in shell scripts.

```
# 'reports' directory was already created in an earlier example
$ mkdir reports
mkdir: cannot create directory ‘reports’: File exists
# exit status will reflect that something went wrong
$ echo $?
1
```

```
# the -p option will override such errors
$ mkdir -p reports
$ echo $?
0
```

As seen in the examples above, you can check the exit status of the last executed command using the `$?` special variable. `0` means everything went well and higher numbers indicate some sort of failure has occurred (the details of which you can look up in the command's manual).

Linux filenames can use any character other than `/` and the ASCII NUL character. Quote the arguments if it contains characters like space, `*`, etc to prevent shell expansion. Shell considers space as the argument separator, `*` is a wildcard character and so on. As a good practice, use only alphabets, numbers and underscores for filenames, unless you have some specific requirements. See also unix.stackexchange: Characters best avoided in filenames.

You can delete the practice directory if you wish:

```
$ cd ..
$ rm -r practice_mkdir
```

touch

You'll usually create files using a text editor or by redirecting the output of a command to a file. For some cases, empty files are needed for testing purposes or to satisfy a particular build process. A real world use case is the empty `.nojekyll` file for GitHub Pages.

The `touch` command's main functionality is altering timestamps (which will be discussed in the File Properties chapter). If a file doesn't exist, `touch` will create an empty file using the current timestamp. You can also pass more than one file argument if needed.

```
$ mkdir practice_touch
$ cd practice_touch

$ ls ip.txt
ls: cannot access 'ip.txt': No such file or directory

$ touch ip.txt

$ ls -s ip.txt
0 ip.txt
```

You can create an empty file using `> ip.txt` as well, but the redirection operator will overwrite the file if it already exists.

rm

The `rm` command will help you to delete files and directories. You can pass one or more paths as arguments.

```
# change to the 'scripts' directory and source the 'rm.sh' script
$ source rm.sh
$ ls -F
empty_dir/  hello.py  loops.py  projects/  read_only.txt  reports/

# delete files ending with .py
$ rm *.py
$ ls -F
empty_dir/  projects/  read_only.txt  reports/
```

You'll need to add the `-r` option to recursively delete directory contents. You can use `rm -d` or the `rmdir` command to delete only empty directories.

```
# -r is needed to delete directory contents recursively
$ rm reports
rm: cannot remove 'reports': Is a directory
$ rm -r reports
$ ls -F
empty_dir/  projects/  read_only.txt

# delete empty directories, same as using the 'rmdir' command
$ rm -d empty_dir
# you'll get an error if the directory is not empty
$ rm -d projects
rm: cannot remove 'projects': Directory not empty
```

Typos like misplaced space, wrong glob, etc could wipe out files not intended for deletion. Apart from having backups and snapshots, you could also take some mitigating steps:

- using `-i` option to interactively delete each file
    - you can also use `-I` option for lesser number of prompts
- using `echo` as a dry run to see how the glob expands
- using a trash command (see links below) instead of `rm`

Use `y` for confirmation and `n` to cancel deletion with the `-i` or `-I` options. Here's an example of cancelling deletion:

```
$ rm -ri projects
rm: descend into directory 'projects'? n

$ ls -F
projects/  read_only.txt
```

And here's an example of providing confirmation at each step of the deletion process:

```
$ tree projects
projects
├── calculator
│   └── calc.sh
└── tictactoe
```

```
    └── game.py

2 directories, 2 files

$ rm -ri projects
rm: descend into directory 'projects'? y
rm: descend into directory 'projects/tictactoe'? y
rm: remove regular empty file 'projects/tictactoe/game.py'? y
rm: remove directory 'projects/tictactoe'? y
rm: descend into directory 'projects/calculator'? y
rm: remove regular empty file 'projects/calculator/calc.sh'? y
rm: remove directory 'projects/calculator'? y
rm: remove directory 'projects'? y

$ ls -F
read_only.txt
```

The `-f` option can be used to ignore complaints about non-existing files (somewhat similar to the `mkdir -p` feature). It also helps to remove write protected files (provided you have appropriate permissions to delete those files). This option is especially useful for recursive deletion of directories that have write protected files, `.git/objects` for example.

```
$ rm xyz.txt
rm: cannot remove 'xyz.txt': No such file or directory
$ echo $?
1
$ rm -f xyz.txt
$ echo $?
0

# example for removing write protected files
# you'll be asked for confirmation even without the -i/-I options
$ rm read_only.txt
rm: remove write-protected regular empty file 'read_only.txt'? n
# with -f, files will be deleted without asking for confirmation
$ rm -f read_only.txt
```

Further Reading

- Use a trash command (for example, `trash-cli` on Ubuntu) so that deleted files can be recovered later if needed
  - see also unix.stackexchange: creating a simple trash command
- Files removed using `rm` can still be recovered with time and skill
  - unix.stackexchange: recover deleted files
  - unix.stackexchange: recovering accidentally deleted files
- Use commands like `shred` if you want to make it harder to recover deleted files
  - wiki.archlinux: Securely wipe disk
- My curated list for `git` and related resources

cp

You can use the `cp` command to make copies of files and directories. With default syntax, you have to specify the source first followed by the destination. To copy multiple items, the last argument as destination can only be a directory. You'll also need to use the `-r` option to copy directories (similar to `rm -r` seen earlier).

```
# change to the 'scripts' directory and source the 'cp.sh' script
$ source cp.sh
$ ls -F
backups/  reference/

# recall that . is a relative path referring to the current directory
$ cp /usr/share/dict/words .
$ ls -F
backups/  reference/  words

# error because -r is needed to copy directories
# other file arguments (if present) will still be copied
$ cp /usr/share/dict .
cp: -r not specified; omitting directory '/usr/share/dict'
$ cp -r /usr/share/dict .
$ ls -F
backups/  dict/  reference/  words
```

 By default, `cp` will overwrite an existing file of the same name in the destination directory. You can use the `-i` option to interactively confirm or deny overwriting existing files. The `-n` option will prevent overwriting existing files without asking for confirmation.

```
$ echo 'hello' > ip.txt
$ ls -F
backups/  dict/  ip.txt  reference/  words
$ ls backups
ip.txt  reports
$ cat backups/ip.txt
apple banana cherry
# file will be overwritten without asking for confirmation!
$ cp ip.txt backups/
$ cat backups/ip.txt
hello

# use -i to interactively confirm or deny overwriting
$ echo 'good morning' > ip.txt
$ cp -i ip.txt backups/
cp: overwrite 'backups/ip.txt'? n
$ cat backups/ip.txt
hello

# use -n to prevent overwriting without needing confirmation
```

```
$ cp -n ip.txt backups/
$ cat backups/ip.txt
hello
```

If there's a folder in the destination path with the same name as a folder being copied, the contents will be merged. If there are files of identical names in such directories, the same rules discussed above will apply.

```
$ tree backups
backups
├── ip.txt
└── reports
    └── jan.log

1 directory, 2 files

$ mkdir reports
$ touch reports/dec.log
$ cp -r reports backups/
$ tree backups
backups
├── ip.txt
└── reports
    ├── dec.log
    └── jan.log

1 directory, 3 files
```

Often, you'd want to copy a file (or a directory) under a different name. In such cases, you can simply use a new name while specifying the destination.

```
# copy 'words' file from source as 'words_ref.txt' at destination
$ cp /usr/share/dict/words words_ref.txt

# copy 'words' file as 'words.txt' under the 'reference' directory
$ cp /usr/share/dict/words reference/words.txt

# copy 'dict' directory as 'word_lists'
$ cp -r /usr/share/dict word_lists
```

As mentioned earlier, to copy multiple files and directories, you'll have to specify the destination directory as the last argument.

```
$ cp -r ~/.bashrc /usr/share/dict backups/

$ ls -AF backups
.bashrc  dict/  ip.txt  reports/
```

You can use the `-t` option to specify the destination before the source paths (helpful with the `find` command for example, will be discussed later). Here are some more notable options:

- `-u` copy files from source only if they are newer or don't exist in the destination
- `-b` and `--backup` options will allow you to create backup copies of files already existing

in the destination
- `--preserve` option will help you to copy files along with source file attributes like ownership, timestamp, etc

Further Reading

- `rsync` a fast, versatile, remote (and local) file-copying tool
  - rsync tutorial and examples
- syncthing — continuous file synchronization program

mv

You can use the `mv` command to move one or more files and directories from one location to another. Unlike `rm` and `cp`, you do not need the `-r` option for directories.

Syntax for specifying the source and destination is same as seen earlier with `cp`. Here's an example of moving a directory into another directory:

```
# change to the 'scripts' directory and source the 'mv.sh' script
$ source mv.sh
$ ls -F
backups/  dot_files/  hello.py  ip.txt  loops.py  manuals/
$ ls -F backups
projects/

$ mv dot_files backups

$ ls -F
backups/  hello.py  ip.txt  loops.py  manuals/
$ ls -F backups
dot_files/  projects/
```

Here's an example for moving multiple files and directories to another directory:

```
$ mv *.py manuals backups

$ ls -F
backups/  ip.txt
$ ls -F backups
dot_files/  hello.py  loops.py  manuals/  projects/
```

When you are dealing with a single file or directory, you can also *rename* them:

```
# within the same directory
$ mv ip.txt report.txt
$ ls -F
backups/  report.txt

# between different directories
$ mv backups/dot_files rc_files
$ ls -F
backups/  rc_files/  report.txt
$ ls -F backups
hello.py  loops.py  manuals/  projects/
```

Here are some more notable options, some of which behave similar to those seen with the `cp` command:

- `-i` interactively confirm or deny when the destination already has a file of the same name
- `-n` always deny overwriting of files
- `-f` always overwrite files
- `-t` specify the destination elsewhere instead of final argument
- `-u` move only if the files are newer or don't exist in the destination
- `-b` and `--backup` options will allow you to create backup copies of files already existing in the destination
- `-v` verbose option

rename

The `mv` command is useful for simple file renaming. `rename` helps when you need to modify one or more filenames based on a pattern. There are different implementations of the `rename` command, with wildly different set of features. See askubuntu: What's the difference between the different "rename" commands? for details.

Perl implementation of the `rename` command will be discussed in this section. You'd need to know regular expressions to use this command. Basic explanations will be given here and more details can be found in the links mentioned at the end of this section. Here's an example to change the file extensions:

```
$ mkdir practice_rename
$ cd practice_rename
# create sample files
$ touch caves.jpeg waterfall.JPEG flower.JPG

# substitution command syntax is s/search/replace/flags
# \. matches . character literally
# e? matches e optionally (? is a quantifier to match 0 or 1 times)
# $ anchors the match to the end of the input
# i flag matches the input case-insensitively
$ rename 's/\.jpe?g$/.jpg/i' *

$ ls
caves.jpg  flower.jpg  waterfall.jpg
$ rm *.jpg
```

As a good practice, use the `-n` option to see how the files will be renamed before actually renaming the files.

```
$ touch 1.png 3.png 25.png 100.png
$ ls
100.png  1.png  25.png  3.png

# use the -n option for sanity check
# note that 100.png isn't part of the output, since it isn't affected
# \d matches a digit character
# \d+ matches 1 or more digits (+ is a quantifier to match 1 or more times)
# e flag treats the replacement string as Perl code
# $& is a backreference to the entire matched portion
```

```
$ rename -n 's/\d+/sprintf "%03d", $&/e' *.png
rename(1.png, 001.png)
rename(25.png, 025.png)
rename(3.png, 003.png)

# remove the -n option after sanity check to actually rename the files
$ rename 's/\d+/sprintf "%03d", $&/e' *.png
$ ls
001.png  003.png  025.png  100.png
```

If the new filename already exists, you'll get an error, which you can override with the `-f` option if you wish. If you are passing filenames with path components in them, you can use the `-d` option to affect only the filename portion. Otherwise, the logic you are using might affect directory names as well.

```
$ mkdir projects
$ touch projects/toc.sh projects/reports.py

# aim is to uppercase the non-extension part of the filename
# [^.]+ matches 1 or more non '.' characters
# \U changes the characters that follow to uppercase
# $& is a backreference to the entire matched portion
$ rename -n -d 's/[^.]+/\U$&/' projects/*
rename(projects/reports.py, projects/REPORTS.py)
rename(projects/toc.sh, projects/TOC.sh)

# without the -d option, directory name will also be affected
$ rename -n 's/[^.]+/\U$&/' projects/*
rename(projects/reports.py, PROJECTS/REPORTS.py)
rename(projects/toc.sh, PROJECTS/TOC.sh)
```

Further Reading

- perldoc: Regexp tutorial
- See my Perl one-liners ebook for examples and more details about the Perl substitution and `rename` commands

ln

The `ln` command helps you create a link to another file or directory within the same or different location. There are two types of links — **symbolic** links and **hard** links. Symbolic links can point to both files and directories. Here are some characteristics:

- if the original file is deleted or moved to another location, then the symbolic link will no longer work
- if the symbolic link is moved to another location, it will still work if the link was done using absolute path (for relative path, it will depend on whether or not there's another file with the same name in that location)
- a symbolic link file has its own inode, permissions, timestamps, etc
- some commands will work the same when original file or the symbolic file is given as the command line argument, while some require additional options (`du -L` for example)

Usage is similar to the `cp` command. You have to specify the source first followed by the destination (which is optional if it is the current working directory).

```
$ mkdir practice_ln
$ cd practice_ln

# destination is optional for making a link in the current directory
# -s option is needed to make symbolic links
$ ln -s /usr/share/dict/words

# you can also rename the link if needed
$ ln -s /usr/share/dict/words words.txt
$ ls -1sF
total 0
0 words@
0 words.txt@
```

Long listing with `ls -l` will show the path connected to links. You can also use the `readlink` command, which has features like resolving recursively to the canonical file.

```
# to know which file the link points to
$ ls -lG words
lrwxrwxrwx 1 learnbyexample 21 Jul  9 13:41 words -> /usr/share/dict/words
$ readlink words
/usr/share/dict/words

# the linked file may be another link
# use -f option to get the original file
$ readlink -f words
/usr/share/dict/english
```

Hard links can only point to another file. You cannot use them for directories and the usage is also restricted to within the same filesystem. The `.` and `..` directories are exceptions, these special purpose hard links are automatically created. Here are some more details about hard links:

- once a hard link is created, there is no distinction between the two files other than their paths. They have same inode, permissions, timestamps, etc
- hard links will continue working even if all the other hard links are deleted
- if a hard link is moved to another location, the links will still be in sync. Any change in one of them will be reflected in all the other links

```
$ touch apple.txt
$ ln apple.txt banana.txt

# the -i option gives inode
$ ls -1i apple.txt banana.txt
649140 banana.txt
649140 apple.txt
```

You can use `unlink` or `rm` commands to delete links.

Further Reading

- askubuntu: What is the difference between a hard link and a symbolic link?
- unix.stackexchange: What is the difference between symbolic and hard links?
- unix.stackexchange: What is a Superblock, Inode, Dentry and a File?

tar and gzip

`tar` is an archiving utility. Depending on the implementation, you can also use options to compress the archive.

Here's an example that creates a single archive file from multiple input files and directories:

```
# change to the 'scripts' directory and source the 'tar.sh' script
$ source tar.sh
$ ls -F
projects/  report.log  todos/

# -c option creates a new archive, any existing archive will be overwritten
# -f option allows to specify a name for the archive being created
# rest of the arguments are the files/directories to be archived
$ tar -cf bkp.tar report.log projects

$ ls -F
bkp.tar  projects/  report.log  todos/
$ ls -sh bkp.tar
7.4M bkp.tar
```

Once you have an archive file, you can then compress it using tools like `gzip`, `bzip2`, `xz`, etc. In the below example, the command replaces the archive file with the compressed version and adds a `.gz` suffix to indicate that `gzip` was the technique used.

```
# the input '.tar' file will be overwritten with the compressed version
$ gzip bkp.tar

$ ls -F
bkp.tar.gz  projects/  report.log  todos/
$ ls -sh bkp.tar.gz
5.6M bkp.tar.gz
```

Use the `-t` option if you want to check the contents of the compressed file. This will work with the uncompressed `.tar` version as well.

```
$ tar -tf bkp.tar.gz
report.log
projects/
projects/scripts/
projects/scripts/calc.sh
projects/errors.log
```

To uncompress `.gz` files, you can use `gunzip` or `gzip -d`. This will replace the compressed version with the uncompressed archive file:

```
# this '.gz' file will be overwritten with the uncompressed version
$ gunzip bkp.tar.gz

$ ls -F
bkp.tar  projects/  report.log  todos/
$ ls -sh bkp.tar
7.4M bkp.tar
```

To extract the files from an archive, use `tar` along with the `-x` option:

```
$ mkdir test_extract
$ mv bkp.tar test_extract
$ cd test_extract
$ ls
bkp.tar

$ tar -xf bkp.tar
$ tree
.
├── bkp.tar
├── projects
│   ├── errors.log
│   └── scripts
│       └── calc.sh
└── report.log

2 directories, 4 files

$ cd ..
$ rm -r test_extract
```

With `GNU tar`, you can compress/uncompress along with the `tar` command instead of having to use tools like `gzip` separately. For example, the `-z` option will use `gzip`, `-j` will use `bzip2` and `-J` will use `xz`. Use the `-a` option if you want `tar` to automatically select the compression technique based on the extension provided.

```
$ ls -F
projects/  report.log  todos/

# -z option gives same compression as the gzip command
$ tar -zcf bkp.tar.gz report.log projects
$ ls -sh bkp.tar.gz
5.6M bkp.tar.gz

# extract original files from compressed file
$ mkdir test_extract
$ cd test_extract
$ tar -zxf ../bkp.tar.gz
$ tree
.
├── projects
```

```
│   ├── errors.log
│   └── scripts
│       └── calc.sh
└── report.log

2 directories, 3 files

$ cd ..
$ rm -r test_extract
```

`tar` has lots and lots of options for various needs. Some are listed below, see documentation for complete details.

- `-v` verbose option
- `-r` to append files to an existing archive
- `--exclude=` specify files to be ignored from archiving

There are also commands starting with `z` to work with compressed files, for example:

- `zcat` to display file contents of a compressed file
- `zless` to display file contents of a compressed file one screenful at a time
- `zgrep` to search compressed files

 If you need to work with `.zip` files, use the `zip` and `unzip` commands.

Further Reading

- unix.stackexchange: tar files with a sorted order
- superuser: gzip without tar? Why are they used together?
- unix.stackexchange: xz a directory with tar using maximum compression?

Exercises

 The `ls.sh` script will be used for some of the exercises.

1) Which of these commands will always display the absolute path of the home directory?

a) `pwd`
b) `echo "$PWD"`
c) `echo "$HOME"`

2) The current working directory has a folder named `-dash`. How would you switch to that directory?

a) `cd -- -dash`
b) `cd -dash`
c) `cd ./-dash`
d) `cd \-dash`
e) `cd '-dash'`

f) all of the above
g) only *a)* and *c)*

3) Given the directory structure as shown below, how would you change to the `todos` directory?

```
# change to the 'scripts' directory and source the 'ls.sh' script
$ source ls.sh

$ ls -F
backups/    hello_world.py*  ip.txt     report.log  todos/
errors.log  hi*              projects/  scripts@
$ cd projects
$ pwd
/home/learnbyexample/cli-computing/example_files/scripts/ls_examples/projects

# ???
$ pwd
/home/learnbyexample/cli-computing/example_files/scripts/ls_examples/todos
```

4) As per the scenario shown below, how would you change to the `cli-computing` directory under the user's home directory? And then, how would you go back to the previous working directory?

```
$ pwd
/home/learnbyexample/all/projects/square_tictactoe

# ???
$ pwd
/home/learnbyexample/cli-computing

# ???
$ pwd
/home/learnbyexample/all/projects/square_tictactoe
```

5) How'd you list the contents of the current directory, one per line, along with the size of the entries in human readable format?

```
# change to the 'scripts' directory and source the 'ls.sh' script
$ source ls.sh

# ???
total 7.4M
4.0K backups
 16K errors.log
4.0K hello_world.py
4.0K hi
4.0K ip.txt
4.0K projects
7.4M report.log
   0 scripts
4.0K todos
```

6) Which `ls` command option would you use for version based sorting of entries?

7) Which `ls` command option would you use for sorting based on entry size?

8) Which `ls` command option would you use for sorting based on file extension?

9) What does the `-G` option of `ls` command do?

10) What does the `-i` option of `ls` command do?

11) List only the directories as one entry per line.

```
# change to the 'scripts' directory and source the 'ls.sh' script
$ source ls.sh

# ???
backups/
projects/
scripts/
todos/
```

12) Assume that a regular file named `notes` already exists. What would happen if you use the `mkdir -p notes` command?

```
$ ls -1F notes
notes

# what would happen here?
$ mkdir -p notes
```

13) Use one or more commands to match the scenario shown below:

```
$ ls -1F
cost.txt

# ???

$ ls -1F
cost.txt
ghost/
quest/
toast/
```

14) Use one or more commands to match the scenario shown below:

```
# start with an empty directory
$ ls -l
total 0

# ???

$ tree -F
.
├── hobbies/
│   ├── painting/
│   │   └── waterfall.bmp
│   ├── trekking/
```

```
|   |   └── himalayas.txt
|   └── writing/
└── shopping/
    └── festival.xlsx

5 directories, 3 files
```

Don't delete this directory, will be needed in a later exercise.

15) If directories to create already exist, which `mkdir` command option would you use to not show an error?

16) Use one or more commands to match the scenario given below:

```
$ ls -1F
cost.txt
ghost/
quest/
toast/

# ???

$ ls -1F
quest/
```

17) What does the `-f` option of `rm` command do?

18) Which option would you use to interactively delete files using the `rm` command?

19) Can the files removed by `rm` easily be restored? Do you need to take some extra steps or use special commands to make the files more difficult to recover?

20) Does your Linux distribution provide a tool to send deleted files to the trash (which would help to recover deleted files)?

21) Which option would you use to interactively accept/prevent the `cp` command from overwriting a file of the same name? And which option would prevent overwriting without needing manual confirmation?

22) Does the `cp` command allow you to rename the file or directory being copied? If so, can you rename multiple files/directories being copied?

23) What do the `-u` , `-b` and `-t` options of `cp` command do?

24) What's the difference between the two commands shown below?

```
$ cp ip.txt op.txt

$ mv ip.txt op.txt
```

25) Which option would you use to interactively accept/prevent the `mv` command from overwriting a file of the same name?

26) Use one or more commands to match the scenario shown below. You should have already created this directory structure in an earlier exercise.

```
$ tree -F
.
├── hobbies/
│   ├── painting/
│   │   └── waterfall.bmp
│   ├── trekking/
│   │   └── himalayas.txt
│   └── writing/
└── shopping/
    └── festival.xlsx

5 directories, 3 files

# ???

$ tree -F
.
├── hobbies/
│   ├── himalayas.txt
│   └── waterfall.bmp
└── shopping/
    └── festival.xlsx

2 directories, 3 files
```

27) What does the `-t` option of `mv` command do?

28) Determine and implement the `rename` logic based on the filenames and expected output shown below.

```
$ touch '(2020) report part 1.txt' 'analysis part 3 (2018).log'
$ ls -1
'(2020) report part 1.txt'
'analysis part 3 (2018).log'

# ???

$ ls -1
2020_report_part_1.txt
analysis_part_3_2018.log
```

29) Does the `ln` command follow the same order to specify source and destination as the `cp` and `mv` commands?

30) Which `tar` option helps to compress archives based on filename extension? This option can be used instead of `-z` for `gzip`, `-j` for `bzip2` and `-J` for `xz`.

Shell Features

This chapter focuses on Bash shell features like quoting mechanisms, wildcards, redirections, command grouping, process substitution, command substitution, etc. Others will be discussed in later chapters.

 The example_files directory has the scripts and sample input files used in this chapter.

 Some of the examples in this chapter use commands that will be discussed in later chapters. Basic description of what such commands do have been added here and you'll also see more examples in the rest of the chapters.

Quoting mechanisms

This section will quote (*heh*) the relevant definitions from the bash manual and provide some examples for each of the four mechanisms.

1) **Escape Character**

A non-quoted backslash `\` is the Bash escape character. It preserves the literal value of the next character that follows, with the exception of newline.
metacharacter: A character that, when unquoted, separates words. A metacharacter is a space, tab, newline, or one of the following characters: `|`, `&`, `;`, `(`, `)`, `<`, or `>`.

Here's an example where unquoted shell metacharacter causes an error:

```
$ echo apple;cherry
apple
cherry: command not found

# '\;' escapes the ';' character, thus losing the metacharacter meaning
$ echo apple\;cherry
apple;cherry
```

And here's an example where the subtler issue might not be apparent at first glance:

```
# this will create two files named 'new' and 'file.txt'
# aim was to create a single file named 'new file.txt'
$ touch new file.txt
$ ls new*txt
ls: cannot access 'new*txt': No such file or directory
$ rm file.txt new

# escaping the space will create a single file named 'new file.txt'
$ touch new\ file.txt
$ ls new*txt
'new file.txt'
$ rm new\ file.txt
```

2) **Single Quotes**

> Enclosing characters in single quotes (`'`) preserves the literal value of each character within the quotes. A single quote may not occur between single quotes, even when preceded by a backslash.

No character is special within single quoted strings. Here's an example:

```
$ echo 'apple;cherry'
apple;cherry
```

You can place strings represented by different quoting mechanisms next to each other to concatenate them together. Here's an example:

```
# concatenation of four strings
# 1: '@fruits = '
# 2: \'
# 3: 'apple and banana'
# 4: \'
$ echo '@fruits = '\''apple and banana'\'
@fruits = 'apple and banana'
```

3) **Double Quotes**

> Enclosing characters in double quotes (`"`) preserves the literal value of all characters within the quotes, with the exception of `$` , `` ` `` , `\` , and, when history expansion is enabled, `!` .

Here's an example showing variable interpolation within double quotes:

```
$ qty='5'

# as seen earlier, no character is special within single quotes
$ echo 'I bought $qty apples'
I bought $qty apples

# a typical use of double quotes is to enable variable interpolation
$ echo "I bought $qty apples"
I bought 5 apples
```

Unless you specifically want the shell to interpret the contents of a variable, you should always quote the variable to avoid issues due to the presence of shell metacharacters.

```
$ f='new file.txt'

# same as: echo 'apple banana' > new file.txt
$ echo 'apple banana' > $f
bash: $f: ambiguous redirect

# same as: echo 'apple banana' > 'new file.txt'
$ echo 'apple banana' > "$f"
$ cat "$f"
```

```
apple banana
$ rm "$f"
```


See also unix.stackexchange: Why does my shell script choke on whitespace or other special characters?.

4) **ANSI-C Quoting**

Words of the form `$'string'` are treated specially. The word expands to string, with backslash-escaped characters replaced as specified by the ANSI C standard.

This form of quoting helps you use escape sequences like `\t` for tab, `\n` for newline and so on. You can also represent characters using their codepoint values in octal and hexadecimal formats.

```
# can also use echo -e 'fig:\t42' or printf 'fig:\t42\n'
$ echo $'fig:\t42'
fig:    42

# \x27 represents the single quote character in hexadecimal format
$ echo $'@fruits = \x27apple and banana\x27'
@fruits = 'apple and banana'

# 'grep' helps you to filter lines based on the given pattern
# but it doesn't recognize escapes like '\t' for tab characters
$ printf 'fig\t42\napple 100\nball\t20\n' | grep '\t'
# in such cases, one workaround is use to ANSI-C quoting
$ printf 'fig\t42\napple 100\nball\t20\n' | grep $'\t'
fig     42
ball    20
```

`printf` is a shell builtin which you can use to format arguments (similar to the `printf()` function from the `C` programming language). This command will be used in many more examples to come.


See bash manual: ANSI-C Quoting for complete list of supported escape sequences. See `man ascii` for a table of ASCII characters and their numerical representations.

Wildcards

It is relatively easy to specify complete filenames as command arguments when they are few in number. And you could use features like tab completion and middle mouse button click (which pastes the last highlighted text) to assist in such cases.

But what to do if you have to deal with tens and hundreds of files (or even more)? If applicable, one way is to match all the files based on a common pattern in their filenames, for example extensions like `.py`, `.txt` and so on. Wildcards (globs) will help in such cases. This feature is provided by the shell, and thus individual commands need not worry about implementing them.

Pattern matching supported by wildcards are somewhat similar to regular expressions, but there are fundamental and syntactical differences between them.

Some of the commonly used wildcards are listed below:

- `*` match any character, zero or more times
  - as a special case, `*` won't match the starting `.` of hidden files unless the `dotglob` shell option is set
- `?` match any character exactly once
- `[set149]` match any of these characters once
- `[^set149]` match any characters *except* the given set of characters
  - you can also use `[!set149]` to negate the character class
- `[a-z]` match a range of characters from `a` to `z`
- `[0-9a-fA-F]` match any hexadecimal character

And here are some examples:

```
# change to the 'scripts' directory and source the 'globs.sh' script
$ source globs.sh
$ ls
100.sh   f1.txt      f4.txt    hi.sh   math.h         report-02.log
42.txt   f2_old.txt  f7.txt    ip.txt  notes.txt      report-04.log
calc.py  f2.txt      hello.py  main.c  report-00.log  report-98.log

# beginning with 'c' or 'h' or 't'
$ ls [cht]*
calc.py  hello.py  hi.sh

# only hidden files and directories
$ ls -d .*
.  ..  .hidden  .somerc

# ending with '.c' or '.py'
$ ls *.c *.py
calc.py  hello.py  main.c

# containing 'o' as well as 'x' or 'y' or 'z' afterwards
$ ls *o*[xyz]*
f2_old.txt  hello.py  notes.txt

# ending with '.' and two more characters
$ ls *.??
100.sh  calc.py  hello.py  hi.sh

# shouldn't start with 'f' and ends with '.txt'
$ ls [^f]*.txt
42.txt  ip.txt  notes.txt

# containing digits '1' to '5' and ending with 'log'
$ ls *[1-5]*log
report-02.log  report-04.log
```

Since some characters are special inside the character class, you need special placement to treat them as ordinary characters:

- `-` should be the first or the last character in the set
- `^` should be other than the first character
- `]` should be the first character

```
$ ls *[ns-]*
100.sh  main.c     report-00.log  report-04.log
hi.sh   notes.txt  report-02.log  report-98.log

$ touch 'a^b' 'mars[planet].txt'
$ rm -i *[]^]*
rm: remove regular empty file 'a^b'? y
rm: remove regular empty file 'mars[planet].txt'? y
```

A **named character set** is defined by a name enclosed between `[:` and `:]` and has to be used within a character class `[]` , along with any other characters as needed.

Named set	Description
`[:digit:]`	`[0-9]`
`[:lower:]`	`[a-z]`
`[:upper:]`	`[A-Z]`
`[:alpha:]`	`[a-zA-Z]`
`[:alnum:]`	`[0-9a-zA-Z]`
`[:word:]`	`[0-9a-zA-Z_]`
`[:xdigit:]`	`[0-9a-fA-F]`
`[:cntrl:]`	control characters — first 32 ASCII characters and 127th (DEL)
`[:punct:]`	all the punctuation characters
`[:graph:]`	`[:alnum:]` and `[:punct:]`
`[:print:]`	`[:alnum:]` , `[:punct:]` and space
`[:ascii:]`	all the ASCII characters
`[:blank:]`	space and tab characters
`[:space:]`	whitespace characters

```
# starting with a digit character, same as: [0-9]*
$ ls [[:digit:]]*
100.sh  42.txt

# starting with a digit character or 'c'
# same as: [0-9c]*
$ ls [[:digit:]c]*
100.sh  42.txt  calc.py

# starting with a non-alphabet character
$ ls [^[:alpha:]]*
100.sh  42.txt
```

 As mentioned before, you can use `echo` to test how the wildcards will expand before using a command to act upon the matching files. For example, `echo *.txt` before using commands like `rm *.txt`. One difference compared to `ls` is that `echo` will display the wildcard as is instead of showing an error if there's no match.

 See bash manual: Pattern Matching for more details, information on locale stuff and so on.

Brace Expansion

This is not a wildcard feature, you just get expanded strings. Brace expansion has two mechanisms for reducing typing:

- taking out common portions among multiple strings
- generating a range of characters

Say you want to create two files named `test_x.txt` and `test_y.txt`. These two strings have something in common at the start and the end. You can specify the unique portions as comma separated strings within a pair of curly braces and put the common parts around the braces. Multiple braces can be used as needed. Use `echo` for testing purposes.

```
$ mkdir practice_brace
$ cd practice_brace

# same as: touch ip1.txt ip3.txt ip7.txt
$ touch ip{1,3,7}.txt
$ ls ip*txt
ip1.txt  ip3.txt  ip7.txt

# same as: mv ip1.txt ip_a.txt
$ mv ip{1,_a}.txt
$ ls ip*txt
ip3.txt  ip7.txt  ip_a.txt

$ echo adders/{half,full}_adder.v
adders/half_adder.v adders/full_adder.v

$ echo file{0,1}.{txt,log}
file0.txt file0.log file1.txt file1.log

# empty alternate is allowed too
$ echo file{,1}.txt
file.txt file1.txt

# example with nested braces
$ echo file.{txt,log{,.bkp}}
file.txt file.log file.log.bkp
```

To generate a range, specify numbers or single characters separated by `..` and an optional

third argument as the step value. Here are some examples:

```
$ echo {1..4}
1 2 3 4
$ echo {4..1}
4 3 2 1

$ echo {1..2}{a..b}
1a 1b 2a 2b

$ echo file{1..4}.txt
file1.txt file2.txt file3.txt file4.txt

$ echo file{1..10..2}.txt
file1.txt file3.txt file5.txt file7.txt file9.txt

$ echo file_{x..z}.txt
file_x.txt file_y.txt file_z.txt

$ echo {z..j..-3}
z w t q n k

# '0' prefix
$ echo {008..10}
008 009 010
```

If the use of braces doesn't match the expansion syntax, it will be left as is:

```
$ echo file{1}.txt
file{1}.txt

$ echo file{1-4}.txt
file{1-4}.txt
```

Extended and Recursive globs

From `man bash` :

Extended glob	Description
`?(pattern-list)`	Matches zero or one occurrence of the given patterns
`*(pattern-list)`	Matches zero or more occurrences of the given patterns
`+(pattern-list)`	Matches one or more occurrences of the given patterns
`@(pattern-list)`	Matches one of the given patterns
`!(pattern-list)`	Matches anything except one of the given patterns

Extended globs are disabled by default. You can use the `shopt` builtin to set/unset **sh**ell **opt**ions like `extglob` , `globstar` , etc. You can also check what is the current status of such options.

```
$ shopt extglob
extglob         off
```

```
# set extglob
$ shopt -s extglob
$ shopt extglob
extglob         on

# unset extglob
$ shopt -u extglob
$ shopt extglob
extglob         off
```

Here are some examples, assuming `extglob` option has already been set:

```
# change to the 'scripts' directory and source the 'globs.sh' script
$ source globs.sh
$ ls
100.sh   f1.txt      f4.txt    hi.sh   math.h         report-02.log
42.txt   f2_old.txt  f7.txt    ip.txt  notes.txt      report-04.log
calc.py  f2.txt      hello.py  main.c  report-00.log  report-98.log

# one or more digits followed by '.' and then zero or more characters
$ ls +([0-9]).*
100.sh  42.txt

# same as: ls *.c *.sh
$ ls *.@(c|sh)
100.sh  hi.sh  main.c

# not ending with '.txt'
$ ls !(*.txt)
100.sh   hello.py  main.c  report-00.log  report-04.log
calc.py  hi.sh     math.h  report-02.log  report-98.log

# not ending with '.txt' or '.log'
$ ls *.!(txt|log)
100.sh  calc.py  hello.py  hi.sh  main.c  math.h
```

If you enable the `globstar` option, you can recursively match filenames within a specified path.

```
# change to the 'scripts' directory and source the 'ls.sh' script
$ source ls.sh

# with 'find' command (this will be explained in a later chapter)
$ find -name '*.txt'
./todos/books.txt
./todos/outing.txt
./ip.txt

# with 'globstar' enabled
$ shopt -s globstar
$ ls **/*.txt
ip.txt  todos/books.txt  todos/outing.txt
```

```
# another example
$ ls -1 **/*.@(py|html)
backups/bookmarks.html
hello_world.py
projects/tictactoe/game.py
```

Add the `shopt` invocations to `~/.bashrc` if you want these settings applied at terminal startup. This will be discussed in the Shell Customization chapter.

set

The `set` builtin command helps you to set or unset values of shell options and positional parameters. Here are some examples for shell options:

```
# disables logging command history from this point onwards
$ set +o history
# enable history logging
$ set -o history

# use vi-style CLI editing interface
$ set -o vi
# use emacs-style interface, this is usually the default
$ set -o emacs
```

You'll see more examples (for example, `set -x`) in later chapters. See bash manual: Set Builtin for documentation.

Pipelines

The pipe control operator `|` helps you connect the output of a command as the input of another command. This operator vastly reduces the need for temporary intermediate files. As discussed previously in the Unix Philosophy section, command line tools usually specialize in a single task. If you can break down a problem into smaller tasks, the pipe operator will come in handy often. Here are some examples:

```
# change to the 'scripts' directory and source the 'du.sh' script
$ source du.sh

# list of files
$ ls
projects  report.log  todos
# count the number of files
# you can also use: printf '%q\n' * | wc -l
$ ls -q | wc -l
3

# report the size of files/folders in human readable format
# and then sort them based on human readable sizes in ascending order
$ du -sh * | sort -h
```

```
8.0K    todos
48K     projects
7.4M    report.log
```

In the above examples, `ls` and `du` perform their own tasks of displaying list of files and showing file sizes respectively. After that, the `wc` and `sort` commands take care of counting and sorting the lines respectively. In such cases, the pipe operator saves you the trouble of dealing with temporary data.

Note that the `%q` format specifier in `printf` helps you quote the arguments in a way that is recognizable by the shell. The `-q` option for `ls` substitutes nongraphic characters in the filenames with a `?` character. Both of these are workarounds to prevent the counting process from getting sidetracked due to characters like newline in the filenames.

 The pipe control operator `|&` will be discussed later in this chapter.

tee

Sometimes, you might want to display the command output on the terminal as well as require the results for later use. In such cases, you can use the `tee` command:

```
$ du -sh * | tee sizes.log
48K     projects
7.4M    report.log
8.0K    todos

$ cat sizes.log
48K     projects
7.4M    report.log
8.0K    todos

$ rm sizes.log
```

Redirection

From bash manual: Redirections:

> Before a command is executed, its input and output may be *redirected* using a special notation interpreted by the shell. Redirection allows commands' file handles to be duplicated, opened, closed, made to refer to different files, and can change the files the command reads from and writes to. Redirection may also be used to modify file handles in the current shell execution environment.

There are three standard data streams:

- **standard input** (`stdin` — file descriptor 0)
- **standard output** (`stdout` — file descriptor 1)
- **standard error** (`stderr` — file descriptor 2)

Both the standard output and error streams are displayed on the terminal by default. The

`stderr` stream is used when something goes wrong with the command usage. Each of these three streams have a predefined file descriptor as mentioned above. In this section, you'll see how to redirect these three streams.


Redirections can be placed anywhere, but they are usually used at the start or end of a command. For example, the following two commands are equivalent:

```
>op.txt grep 'error' report.log

grep 'error' report.log >op.txt
```


Space characters between the redirection operators and the filename are optional.

Redirecting output

You can use the `>` operator to redirect the standard output of a command to a file. A number prefix can be added to the `>` operator to work with that particular file descriptor. Default is `1` (recall that the file descriptor for `stdout` is `1`), so `1>` and `>` perform the same operation. Use `>>` to append the output to a file.

The filename provided to the `>` and `>>` operators will be created if a regular file of that name doesn't exist yet. If the file already exists, `>` will overwrite that file whereas `>>` will append the contents.

```
# change to the 'example_files/text_files' directory for this section

# save first three lines of 'sample.txt' to 'op.txt'
$ head -n3 sample.txt > op.txt
$ cat op.txt
 1) Hello World
 2)
 3) Hi there

# append last two lines of 'sample.txt' to 'op.txt'
$ tail -n2 sample.txt >> op.txt
$ cat op.txt
 1) Hello World
 2)
 3) Hi there
14) He he he
15) Adios amigo

$ rm op.txt
```


You can use `/dev/null` as a filename to discard the output, to provide an empty file as input for a command, etc.

You can use `set noclobber` to prevent overwriting if a file already exists. When the `noclobber` option is set, you can still overwrite a file by using `>|` instead of the `>` operator.

Redirecting input

Some commands like `tr` and `datamash` can only work with data from the standard input. This isn't an issue when you are piping data from another command, for example:

```
# filter lines containing 'the' from the input file 'greeting.txt'
# and then display the results in uppercase using the 'tr' command
$ grep 'the' greeting.txt | tr 'a-z' 'A-Z'
HI THERE
```

You can use the `<` redirection operator if you want to pass data from a file to such commands. The default prefix here is `0`, which is the file descriptor for `stdin` data. Here's an example:

```
$ tr 'a-z' 'A-Z' <greeting.txt
HI THERE
HAVE A NICE DAY
```

In some cases, a tool behaves differently when processing `stdin` data compared to file input. Here's an example with `wc -l` to report the total number of lines in the input:

```
# line count, filename is part of the output as well
$ wc -l purchases.txt
8 purchases.txt

# filename won't be part of the output for stdin data
# helpful for assigning the number to a variable for scripting purposes
$ wc -l <purchases.txt
8
```

Sometimes, you need to pass `stdin` data as well as other file inputs to a command. In such cases, you can use `-` to represent data from the standard input. Here's an example:

```
$ cat scores.csv
Name,Maths,Physics,Chemistry
Ith,100,100,100
Cy,97,98,95
Lin,78,83,80

# insert a column at the start
$ printf 'ID\n1\n2\n3' | paste -d, - scores.csv
ID,Name,Maths,Physics,Chemistry
1,Ith,100,100,100
2,Cy,97,98,95
3,Lin,78,83,80
```

Even though a command accepts file input directly as an argument, redirecting can help for interactive usage. Here's an example:

```
# display only the third field
$ <scores.csv cut -d, -f3
Physics
100
98
83

# later, you realize that you need the first field too
# use 'up' arrow key to bring the previous command
# and modify the argument easily at the end
# if you had used cut -d, -f3 scores.csv instead,
# you'd have to navigate past the filename to modify the argument
$ <scores.csv cut -d, -f1,3
Name,Physics
Ith,100
Cy,98
Lin,83
```

 Don't use `cat filename | cmd` for passing file content as `stdin` data, unless you need to concatenate data from multiple input files. See wikipedia: UUOC and Useless Use of Cat Award for more details.

Redirecting error

Recall that the file descriptor for `stderr` is `2`. So, you can use `2>` to redirect standard error to a file. Use `2>>` if you need to append the contents. Here's an example:

```
# assume 'abcdxyz' doesn't exist as a shell command
$ abcdxyz
abcdxyz: command not found

# the error in such cases will be part of the stderr stream, not stdout
# so, you'll need to use 2> here
$ abcdxyz 2> cmderror.log
$ cat cmderror.log
abcdxyz: command not found

$ rm cmderror.log
```

 Use `/dev/null` as a filename if you need to discard the results.

Combining stdout and stderr

Newer versions of Bash provide these handy shortcuts:

- `&>` redirect both `stdout` and `stderr` (overwrite if file already exists)
- `&>>` redirect both `stdout` and `stderr` (append if file already exists)
- `|&` pipe both `stdout` and `stderr` as input to another command

Here's an example which assumes `xyz.txt` doesn't exist, thus leading to errors:

```
# using '>' will redirect only the stdout stream
# stderr will be displayed on the terminal
$ grep 'log' file_size.txt xyz.txt > op.txt
grep: xyz.txt: No such file or directory

# using '&>' will redirect both the stdout and stderr streams
$ grep 'log' file_size.txt xyz.txt &> op.txt
$ cat op.txt
file_size.txt:104K    power.log
file_size.txt:746K    report.log
grep: xyz.txt: No such file or directory

$ rm op.txt
```

And here's an example with the `|&` operator:

```
# filter lines containing 'log' from the given file arguments
# and then filter lines containing 'or' from the combined stdout and stderr
$ grep 'log' file_size.txt xyz.txt |& grep 'or'
file_size.txt:746K    report.log
grep: xyz.txt: No such file or directory
```

For earlier Bash versions, you'll have to manually redirect the streams:

- `1>&2` redirects file descriptor `1` (`stdout`) to the file descriptor `2` (`stderr`)
- `2>&1` redirects file descriptor `2` (`stderr`) to the file descriptor `1` (`stdout`)

Here are some examples:

```
# note that the order of redirections is important here
# you can also use: 2> op.txt 1>&2
$ grep 'log' file_size.txt xyz.txt > op.txt 2>&1
$ cat op.txt
file_size.txt:104K    power.log
file_size.txt:746K    report.log
grep: xyz.txt: No such file or directory
$ rm op.txt

$ grep 'log' file_size.txt xyz.txt 2>&1 | grep 'or'
file_size.txt:746K    report.log
grep: xyz.txt: No such file or directory
```

Waiting for stdin

Sometimes, you might mistype a command without providing input. And instead of getting an error, you'll see the cursor patiently waiting for something. This isn't the shell hanging up on you. The command is waiting for you to type data, so that it can perform its task.

Say, you typed `cat` and pressed the Enter key. Seeing the blinking cursor, you type some text and press the Enter key again. You'll see the text you just typed echoed back to you as `stdout` (which is the functionality of the `cat` command). This will continue again and again, until you tell the shell that you are done. How to do that? Press `Ctrl+d` on a fresh line or press `Ctrl+d`

twice at the end of a line. In the latter case, you'll not get a newline character at the end of the data.

```
# press Enter and Ctrl+d after typing all the required characters
$ cat
knock knock
knock knock
anybody here?
anybody here?

# 'tr' command here translates lowercase to uppercase
$ tr 'a-z' 'A-Z'
knock knock
KNOCK KNOCK
anybody here?
ANYBODY HERE?
```

Getting output immediately after each input line depends on the command's functionality. Commands like `sort` and `shuf` will wait for the entire input data before producing the output.

```
# press Ctrl+d after the third input line
$ sort
lion
zebra
bee
bee
lion
zebra
```

Here's an example which has output redirection as well:

```
# press Ctrl+d after the line containing 'histogram'
# filter lines containing 'is'
$ grep 'is' > op.txt
hi there
this is a sample line
have a nice day
histogram

$ cat op.txt
this is a sample line
histogram

$ rm op.txt
```

See also unix.stackexchange: difference between Ctrl+c and Ctrl+d.

Here Documents

Here Documents is another way to provide `stdin` data. In this case, the termination condition is a line matching a predefined string which is specified after the `<<` redirection operator. This is especially helpful for automation, since pressing `Ctrl+d` interactively isn't desirable. Here's an example:

```
# EOF is typically used as the special string
$ cat << 'EOF' > fruits.txt
> banana 2
> papaya 3
> mango  10
> EOF

$ cat fruits.txt
banana 2
papaya 3
mango  10
$ rm fruits.txt
```

In the above example, the termination string was enclosed in single quotes as a good practice. Doing so prevents parameter expansion, command substitution, etc. You can also use `\string` for this purpose. If you use `<<-` instead of `<<` , leading tab characters can be added at the start of input lines without being part of the actual data.

Just like `$` and a space represents the primary prompt (`PS1` shell variable), `>` and a space at the start of lines represents the secondary prompt `PS2` (applicable for multiline commands). Don't type these characters when you use Here Documents in a shell script.

See bash manual: Here Documents and stackoverflow: here documents for more examples and details.

Here Strings

This is similar to Here Documents, but the string is passed as an argument after the `<<<` redirection operator. Here are some examples:

```
$ tr 'a-z' 'A-Z' <<< hello
HELLO
$ tr 'a-z' 'A-Z' <<< 'hello world'
HELLO WORLD

$ greeting='hello world'
$ tr 'a-z' 'A-Z' > op.txt <<< "$greeting"
$ cat op.txt
HELLO WORLD
$ rm op.txt
```

Further Reading

- Short introduction to shell redirection
- Illustrated Redirection Tutorial
- stackoverflow: Redirect a stream to another file descriptor using >&
- Difference between 2>&1 >foo and >foo 2>&1
- stackoverflow: Redirect and append both stdout and stderr to a file
- unix.stackexchange: Examples for <> redirection

Grouping commands

You can use the `(list)` and `{ list; }` compound commands to redirect content for several commands. The former is executed in a subshell whereas the latter is executed in the current shell context. Spaces around `()` are optional but necessary for the `{}` version. From bash manual: Lists of Commands:

> A `list` is a sequence of one or more pipelines separated by one of the operators `;` , `&` , `&&` , or `||` , and optionally terminated by one of `;` , `&` , or a newline.

Here are some examples of command groupings:

```
# change to the 'example_files/text_files' directory for this section

# the 'sed' command here gives the first line of the input
# rest of the lines are then processed by the 'sort' command
# thus, the header will always be the first line in the output
$ (sed -u '1q' ; sort) < scores.csv
Name,Maths,Physics,Chemistry
Cy,97,98,95
Ith,100,100,100
Lin,78,83,80

# save first three and last two lines from 'sample.txt' to 'op.txt'
$ { head -n3 sample.txt; tail -n2 sample.txt; } > op.txt
$ cat op.txt
 1) Hello World
 2)
 3) Hi there
14) He he he
15) Adios amigo
$ rm op.txt
```

You might wonder why the second command did not use `< sample.txt` instead of repeating the filename twice. The reason is that some commands might read more than what is required (for buffering purposes) and thus cause issues for the remaining commands. In the `sed+sort` example, the `-u` option guarantees that `sed` will not to read more than the required data. See unix.stackexchange: sort but keep header line at the top for more examples and details.

You don't need the `()` or `{}` groups to see the results of multiple commands on the terminal. Just the `;` separator between the commands would be enough. See also bash manual: Command Execution Environment.

```
$ head -n1 sample.txt ; echo 'have a nice day'
 1) Hello World
have a nice day
```

List control operators

You can use these operators to control the execution of the subsequent command depending on the exit status of the first command. From bash manual: Lists of Commands:

> AND and OR lists are sequences of one or more pipelines separated by the control operators `&&` and `||`, respectively. AND and OR lists are executed with left associativity.

For AND list, the second command will be executed if and only if the first command exits with `0` status.

```
# first command succeeds here, so the second command is also executed
$ echo 'hello' && echo 'have a nice day'
hello
have a nice day

# assume 'abcdxyz' doesn't exist as a shell command
# the second command will not be executed
$ abcdxyz && echo 'have a nice day'
abcdxyz: command not found

# if you use ';' instead, the second command will still be executed
$ abcdxyz ; echo 'have a nice day'
abcdxyz: command not found
have a nice day
```

For OR list, the second command will be executed if and only if the first command does *not* exit with `0` status.

```
# since the first command succeeds, the second one won't run
$ echo 'hello' || echo 'have a nice day'
hello

# assume 'abcdxyz' doesn't exist as a shell command
# since the first command fails, the second one will run
$ abcdxyz || echo 'have a nice day'
abcdxyz: command not found
have a nice day
```

Command substitution

Command substitution allows you to use the standard output of a command as part of another command. Trailing newlines, if any, will be removed. You can use the newer and preferred syntax `$(command)` or the older syntax `` `command` ``. Here are some examples:

```
# sample input
$ printf 'hello\ntoday is: \n'
hello
today is:
# append output from the 'date' command to the line containing 'today'
$ printf 'hello\ntoday is: \n' | sed '/today/ s/$/'"$(date +%A)"'/'
hello
today is: Monday

# save the output of 'wc' command to a variable
# same as: line_count=`wc -l <sample.txt`
$ line_count=$(wc -l <sample.txt)
$ echo "$line_count"
15
```

Here's an example with nested substitutions:

```
# dirname removes the trailing path component
$ dirname projects/tictactoe/game.py
projects/tictactoe
# basename removes the leading directory component
$ basename projects/tictactoe
tictactoe

$ proj=$(basename $(dirname projects/tictactoe/game.py))
$ echo "$proj"
tictactoe
```

Difference between the two types of syntax is quoted below from bash manual: Command Substitution:

> When the old-style backquote form of substitution is used, backslash retains its literal meaning except when followed by `$`, `` ` ``, or `\`. The first backquote not preceded by a backslash terminates the command substitution. When using the $(command) form, all characters between the parentheses make up the command; none are treated specially. Command substitutions may be nested. To nest when using the backquoted form, escape the inner backquotes with backslashes.

Process substitution

Instead of a file argument, you can use command output with process substitution. The syntax is `<(list)`. The shell will take care of passing a filename with the standard output of those commands. Here's an example:

```
# change to the 'example_files/text_files' directory for this section

$ cat scores.csv
```

```
Name,Maths,Physics,Chemistry
Ith,100,100,100
Cy,97,98,95
Lin,78,83,80

# can also use: paste -d, <(echo 'ID'; seq 3) scores.csv
$ paste -d, <(printf 'ID\n1\n2\n3') scores.csv
ID,Name,Maths,Physics,Chemistry
1,Ith,100,100,100
2,Cy,97,98,95
3,Lin,78,83,80
```

For the above example, you could also have used `-` to represent `stdin` piped data as seen in an earlier section. Here's an example where two substitutions are used. This essentially helps you to avoid managing multiple temporary files, similar to how the `|` pipe operator helps for single temporary file.

```
# side-by-side view of sample input files
$ paste f1.txt f2.txt
1       1
2       hello
3       3
world   4

# this command gives the common lines between two files
# the files have to be sorted for the command to work properly
$ comm -12 <(sort f1.txt) <(sort f2.txt)
1
3
```

 See this unix.stackexchange thread for examples with the `>(list)` form.

Exercises

 Use the `globs.sh` script for wildcards related exercises, unless otherwise mentioned.

 Create a temporary directory for exercises that may require you to create some files. You can delete such practice directories afterwards.

1) Use the `echo` command to display the text as shown below. Use appropriate quoting as necessary.

```
# ???
that's    great! $x = $y + $z
```

2) Use the `echo` command to display the values of the three variables in the format as shown below.

```
$ n1=10
$ n2=90
$ op=100

# ???
10 + 90 = 100
```

3) What will be the output of the command shown below?

```
$ echo $'\x22apple\x22: \x2710\x27'
```

4) List filenames starting with a digit character.

```
# change to the 'scripts' directory and source the 'globs.sh' script
$ source globs.sh

# ???
100.sh  42.txt
```

5) List filenames whose extension do not begin with `t` or `l`. Assume extensions will have at least one character.

```
# ???
100.sh  calc.py  hello.py  hi.sh  main.c  math.h
```

6) List filenames whose extension only have a single character.

```
# ???
main.c  math.h
```

7) List filenames whose extension is not `txt`.

```
# ???
100.sh   hello.py  main.c  report-00.log  report-04.log
calc.py  hi.sh     math.h  report-02.log  report-98.log
```

8) Describe the wildcard pattern used in the command shown below.

```
$ ls *[^[:word:]]*.*
report-00.log  report-02.log  report-04.log  report-98.log
```

9) List filenames having only lowercase alphabets before the extension.

```
# ???
calc.py  hello.py  hi.sh  ip.txt  main.c  math.h  notes.txt
```

10) List filenames starting with `ma` or `he` or `hi`.

```
# ???
hello.py  hi.sh  main.c  math.h
```

11) What commands would you use to get the outputs shown below? Assume that you do not know the depth of sub-directories.

```
# change to the 'scripts' directory and source the 'ls.sh' script
$ source ls.sh

# filenames ending with '.txt'
```

```
# ???
ip.txt  todos/books.txt  todos/outing.txt

# directories starting with 'c' or 'd' or 'g' or 'r' or 't'
# ???
backups/dot_files/
projects/calculator/
projects/tictactoe/
todos/
```

12) Create and change to an empty directory. Then, use brace expansion along with relevant commands to get the results shown below.

```
# ???
$ ls report*
report_2020.txt  report_2021.txt  report_2022.txt

# use the 'cp' command here
# ???
$ ls report*
report_2020.txt  report_2021.txt  report_2021.txt.bkp  report_2022.txt
```

13) What does the `set` builtin command do?

14) What does the `|` pipe operator do? And when would you add the `tee` command?

15) Can you infer what the following command does? *Hint*: see `help printf` .

```
$ printf '%s\n' apple car dragon
apple
car
dragon
```

16) Use brace expansion along with relevant commands and shell features to get the result shown below. *Hint*: see previous question.

```
$ ls ip.txt
ls: cannot access 'ip.txt': No such file or directory

# ???
$ cat ip.txt
item_10
item_12
item_14
item_16
item_18
item_20
```

17) With `ip.txt` containing text as shown in the previous question, use brace expansion and relevant commands to get the result shown below.

```
# ???
$ cat ip.txt
item_10
```

```
item_12
item_14
item_16
item_18
item_20
apple_1_banana_6
apple_1_banana_7
apple_1_banana_8
apple_2_banana_6
apple_2_banana_7
apple_2_banana_8
apple_3_banana_6
apple_3_banana_7
apple_3_banana_8
```

18) What are the differences between `<` and `|` shell operators, if any?

19) Which character is typically used to represent `stdin` data as a file argument?

20) What do the following operators do?

a) `1>`
b) `2>`
c) `&>`
d) `&>>`
e) `|&`

21) What will be the contents of `op.txt` if you use the following `grep` command?

```
# press Ctrl+d after the line containing 'histogram'
$ grep 'hi' > op.txt
hi there
this is a sample line
have a nice day
histogram

$ cat op.txt
```

22) What will be the contents of `op.txt` if you use the following commands?

```
$ qty=42
$ cat << end > op.txt
> dragon
> unicorn
> apple $qty
> ice cream
> end

$ cat op.txt
```

23) Correct the command to get the expected output shown below.

```
$ books='cradle piranesi soulhome bastion'
```

```
# something is wrong with this command
$ sed 's/\b\w/\u&/g' <<< '$books'
$Books

# ???
Cradle Piranesi Soulhome Bastion
```

24) Correct the command to get the expected output shown below.

```
# something is wrong with this command
$ echo 'hello' ; seq 3 > op.txt
hello
$ cat op.txt
1
2
3

# ???
$ cat op.txt
hello
1
2
3
```

25) What will be the output of the following commands?

```
$ printf 'hello' | tr 'a-z' 'A-Z' && echo ' there'

$ printf 'hello' | tr 'a-z' 'A-Z' || echo ' there'
```

26) Correct the command(s) to get the expected output shown below.

```
# something is wrong with these commands
$ nums=$(seq 3)
$ echo $nums
1 2 3

# ???
1
2
3
```

27) Will the following two commands produce equivalent output? If not, why not?

```
$ paste -d, <(seq 3) <(printf '%s\n' item_{1..3})

$ printf '%s\n' {1..3},item_{1..3}
```

Viewing Part or Whole File Contents

In this chapter, you'll learn how to view contents of files from within the terminal. If the contents are too long, you can choose to view one screenful at a time or get only the starting/ending portions of the input. The commands used for these purposes also have other functionalities, some of which will be discussed in this chapter as well.


The example_files directory has the sample input files used in this chapter.

cat

The `cat` command derives its name from con**cat**enate. It is primarily used to combine the contents of multiple files to be saved in a file or sent as input to another command.

Commonly used options are shown below:

- `-n` prefix line number and a tab character to each input line
- `-b` like `-n` but doesn't number empty lines
- `-s` squeeze consecutive empty lines to a single empty line
- `-v` view special characters like NUL using the caret notation
- `-e` view special characters as well as mark the end of line
- `-A` includes `-e` and also helps to spot tab characters

Here are some examples to showcase `cat` 's main utility. One or more files can be given as arguments.


As mentioned earlier, the example_files directory has the sample input files used in this chapter. You need to `cd` into the `example_files/text_files` directory to follow along the examples shown in this chapter.

```
# view contents of a single file
$ cat greeting.txt
Hi there
Have a nice day

# another example
$ cat fruits.txt
banana
papaya
mango

# concatenate multiple files
$ cat greeting.txt fruits.txt
Hi there
Have a nice day
banana
papaya
mango
```

To save the output of concatenation, use redirection:

```
$ cat greeting.txt fruits.txt > op.txt

$ cat op.txt
Hi there
Have a nice day
banana
papaya
mango
```

You can represent `stdin` data using `-` as a file argument. If file arguments are not present, `cat` will read from `stdin` data if present or wait for interactive input. Note that `-` is also supported by many more commands to indicate `stdin` data.

```
# concatenate contents of 'greeting.txt' and 'stdin' data
$ echo 'apple banana cherry' | cat greeting.txt -
Hi there
Have a nice day
apple banana cherry
```

Using `cat` to view the contents of a file, to concatenate them, etc is well and good. But, using `cat` when it is not needed is a bad habit that you should avoid. See wikipedia: UUOC and Useless Use of Cat Award for more details.

`cat` also helps you spot special characters using the caret notation:

```
# example for backspace and carriage return characters
$ printf 'car\bd\nbike\rp\n'
cad
pike
$ printf 'car\bd\nbike\rp\n' | cat -v
car^Hd
bike^Mp

# example with tab characters and end-of-line marker
$ printf '1 2\t3\f4\v5   \n' | cat -A
1 2^I3^L4^K5   $
```

tac

You can concatenate files using `tac` as well, but the output will be printed in the reverse (line wise). If you pass multiple input files, each file content will be reversed separately. Here are some examples:

```
$ printf 'apple\nbanana\ncherry\n' | tac
cherry
banana
apple

# won't be same as: cat greeting.txt fruits.txt | tac
$ tac greeting.txt fruits.txt
Have a nice day
```

```
Hi there
mango
papaya
banana
```

 If the last line of input doesn't end with a newline, the output will also not have that newline character.

```
$ printf 'apple\nbanana\ncherry' | tac
cherrybanana
apple
```

less

The `cat` command is not suitable for viewing contents of large files in the terminal. The `less` command automatically fits the content to the size of the terminal, allows scrolling and has nifty features for effective viewing. Typically, the `man` command uses `less` as the `pager` to display the documentation. The navigation options are similar to the Vim text editor.

Commonly used commands are given below. You can press the `h` key for builtin help.

- `↑` and `↓` arrow keys to move up and down by a line
  - you can also use `k` and `j` keys (same keys as those used by the Vim text editor)
- `f` and `b` keys to move forward and backward by a screenful of content
  - `Space` key also moves forward by a screen
- mouse scroll moves up and down by a few lines
- `g` or `Home` go to the start of the file
- `G` or `End` go to the end of the file
- `/pattern` followed by `Enter` search for the given pattern in the forward direction
  - pattern refers to regular expressions and depends on the regex library in your system
  - the flavor is Extended Regular Expressions (ERE) on my system
  - see `man re_format` for more details
- `?pattern` followed by `Enter` search for the given pattern in the backward direction
- `n` go to the next match
- `N` go to the previous match
- `q` quit

As an example, use `less /usr/share/dict/words` to open a dictionary file and practice the commands discussed above. If your `pager` is set to `less` for manual pages, you can also try something like `man ls` for practice.

Similar to the `cat` command, you can use the `-s` option to squeeze consecutive blank lines. But unlike `cat -n`, you need to use `less -N` to prefix line numbers. The lowercase `-n` option will turn off numbering.

Further Reading

- `less` command is an improved version of the `more` command
- unix.stackexchange: differences between most, more and less
- My Vim Reference Guide ebook

tail

By default, `tail` displays the last 10 lines of input files. If there are less than 10 lines in the input, only those lines will be displayed. You can use the `-n` option to change the number of lines displayed. By using `tail -n +N`, you can get all the lines starting from the `N` th line.

Here's an example file that'll be used for illustration purposes:

```
$ cat sample.txt
 1) Hello World
 2)
 3) Hi there
 4) How are you
 5)
 6) Just do-it
 7) Believe it
 8)
 9) banana
10) papaya
11) mango
12)
13) Much ado about nothing
14) He he he
15) Adios amigo
```

Here are some examples with the `-n` option:

```
# last two lines (input has 15 lines)
$ tail -n2 sample.txt
14) He he he
15) Adios amigo

# all lines starting from the 11th line
# space between -n and +N is optional
$ tail -n +11 sample.txt
11) mango
12)
13) Much ado about nothing
14) He he he
15) Adios amigo
```

If you pass multiple input files, each file will be processed separately. By default, the output is nicely formatted with filename headers and empty line separators which you can override with the `-q` (quiet) option.

```
$ tail -n2 fruits.txt sample.txt
==> fruits.txt <==
papaya
banana

==> sample.txt <==
14) He he he
15) Adios amigo
```

The `-c` option works similar to the `-n` option, but with bytes instead of lines:

```
# last three bytes
# note that the input doesn't end with a newline character
$ printf 'apple pie' | tail -c3
pie

# starting from the fifth byte
$ printf 'car\njeep\nbus\n' | tail -c +5
jeep
bus
```

Further Reading

- wikipedia: File monitoring with tail -f and -F options
    - toolong — terminal application to view, tail, merge, and search log files
- unix.stackexchange: How does the tail -f option work?
- How to deal with output buffering?

head

By default, `head` displays the first 10 lines of input files. If there are less than 10 lines in the input, only those lines will be displayed. You can use the `-n` option to change the number of lines displayed. By using `head -n -N`, you can get all the input lines except the last `N` lines.

```
# first three lines
$ head -n3 sample.txt
 1) Hello World
 2)
 3) Hi there

# except the last 11 lines
$ head -n -11 sample.txt
 1) Hello World
 2)
 3) Hi there
 4) How are you
```

You can select a range of lines by combining both the `head` and `tail` commands.

```
# 9th to 11th lines
# same as: tail -n +9 sample.txt | head -n3
$ head -n11 sample.txt | tail -n +9
 9) banana
10) papaya
11) mango
```

If you pass multiple input files, each file will be processed separately. By default, the output is nicely formatted with filename headers and empty line separators which you can override with the `-q` (quiet) option.

```
$ printf '1\n2\n' | head -n1 greeting.txt -
==> greeting.txt <==
Hi there
```

```
==> standard input <==
1
```

The `-c` option works similar to the `-n` option, but with bytes instead of lines:

```
# first three bytes
$ printf 'apple pie' | head -c3
app

# excluding the last four bytes
$ printf 'car\njeep\nbus\n' | head -c -4
car
jeep
```

Exercises

Use the example_files/text_files directory for input files used in the following exercises.

1) Which option(s) would you use to get the output shown below?

```
$ printf '\n\n\ndragon\n\n\nunicorn\n\n\n' | cat # ???

     1  dragon

     2  unicorn
```

2) Pass appropriate arguments to the `cat` command to get the output shown below.

```
$ cat greeting.txt
Hi there
Have a nice day

$ echo '42 apples and 100 bananas' | cat # ???
42 apples and 100 bananas
Hi there
Have a nice day
```

3) Will the two commands shown below produce the same output? If not, why not?

```
$ cat fruits.txt ip.txt | tac

$ tac fruits.txt ip.txt
```

4) Go through the manual for the `tac` command and use appropriate options and arguments to get the output shown below.

```
$ cat blocks.txt
%=%=
apple
banana
%=%=
```

```
brown
green

# ???
%=%=
brown
green
%=%=
apple
banana
```

5) What is the difference between `less -n` and `less -N` options? Does `cat -n` and `less -n` have similar functionality?

6) Which command would you use to open another file from within an existing `less` session? And which commands would you use to navigate between previous and next files?

7) Use appropriate commands and shell features to get the output shown below.

```
$ printf 'carpet\njeep\nbus\n'
carpet
jeep
bus

# use the above 'printf' command for input data
$ c=# ???
$ echo "$c"
car
```

8) How would you display all the input lines except the first one?

```
$ printf 'apple\nfig\ncarpet\njeep\nbus\n' | # ???
fig
carpet
jeep
bus
```

9) Which command(s) would you use to get the output shown below?

```
$ cat fruits.txt
banana
papaya
mango
$ cat blocks.txt
%=%=
apple
banana
%=%=
brown
green

# ???
banana
papaya
```

```
%=%=
apple
```

10) Use a combination of the `head` and `tail` commands to get the 11th to 14th characters from the given input.

```
$ printf 'apple\nfig\ncarpet\njeep\nbus\n' | # ???
carp
```

11) Extract the starting six bytes from the input files `table.txt` and `fruits.txt` .

```
# ???
brown banana
```

12) Extract the last six bytes from the input files `fruits.txt` and `table.txt` .

```
# ???
mango
 3.14
```