Just enough command line (Unix & Linux)

Frank Burkholder

Objectives

- Recommend initial file structure for course
- Define an Operating System
- History of Unix & Linux
- Advantages of the Unix philosophy and the command line
- General Unix file structure
- General syntax of Unix commands
- Overview of common commands
- Pipes
- Redirecting output

Recommended organization/file structure

```
~/Documents
    |-dsi
        |-repos
        |-lectures
        |-solutions-den18
```

source

Make this structure from the command line

1. Get to Terminal: Ctrl - shift - t (Linux) or Command - t (Mac)
2. Change into the Documents directory.

```
frank@peregrine:~$ cd ~/Documents
```

3. Make the dsi directory.

```
frank@peregrine:~/Documents$ mkdir dsi
```

4. Change into the dsi directory.

```
frank@peregrine:~/Documents$ cd dsi/
frank@peregrine:~/Documents/dsi$
```

5. Make the repos directory (it's not a git repository). This where you will clone daily git repos.

```
frank@peregrine:~/Documents/dsi$ mkdir repos
```

6. Add the lectures directory (it **is** a git repository, so cloning it; limit depth but include all branches).

```
dsi$ git clone https://github.com/GalvanizeDataScience/lectures.git
```

7. Add the solutions directory (git repository, not large nor many commits so not worried about depth).

```
~/Documents/dsi$ git clone https://github.com/GalvanizeDataScience/solutions-den20.git
```

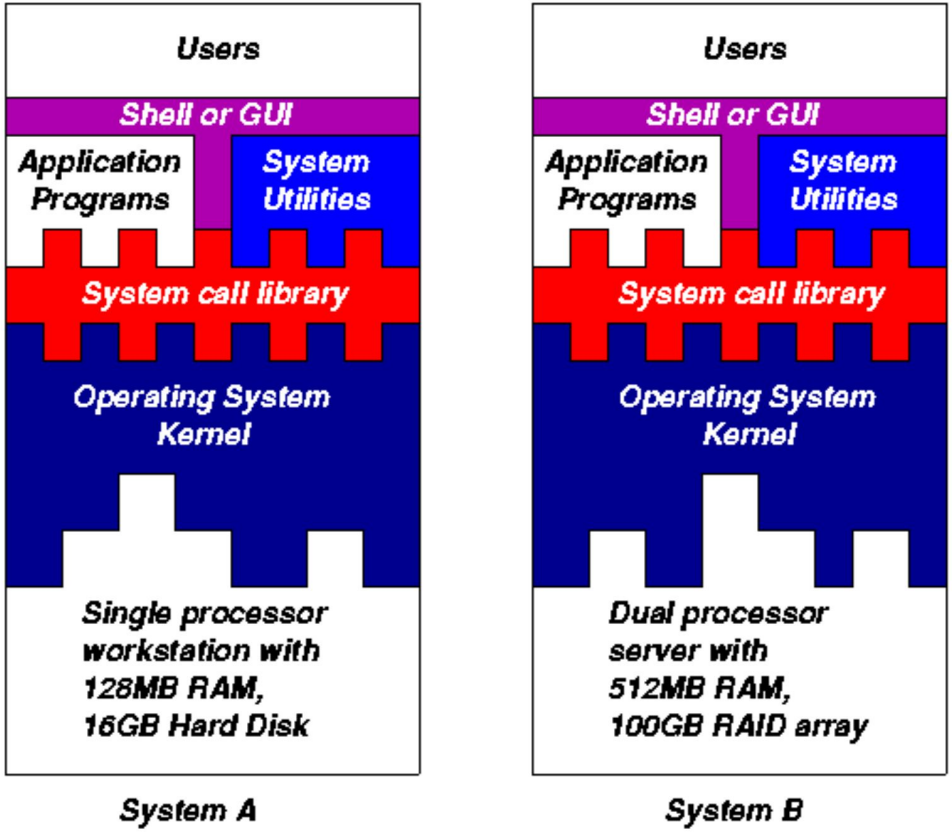
8. Check that all directories are there.

```
(base) frank@peregrine:~/Documents/dsi$ ls
lectures  repos  solutions-den20
```

```
~/Documents
    |-dsi
        |-repos
        |-lectures
        |-solutions-den18
```

Operating system

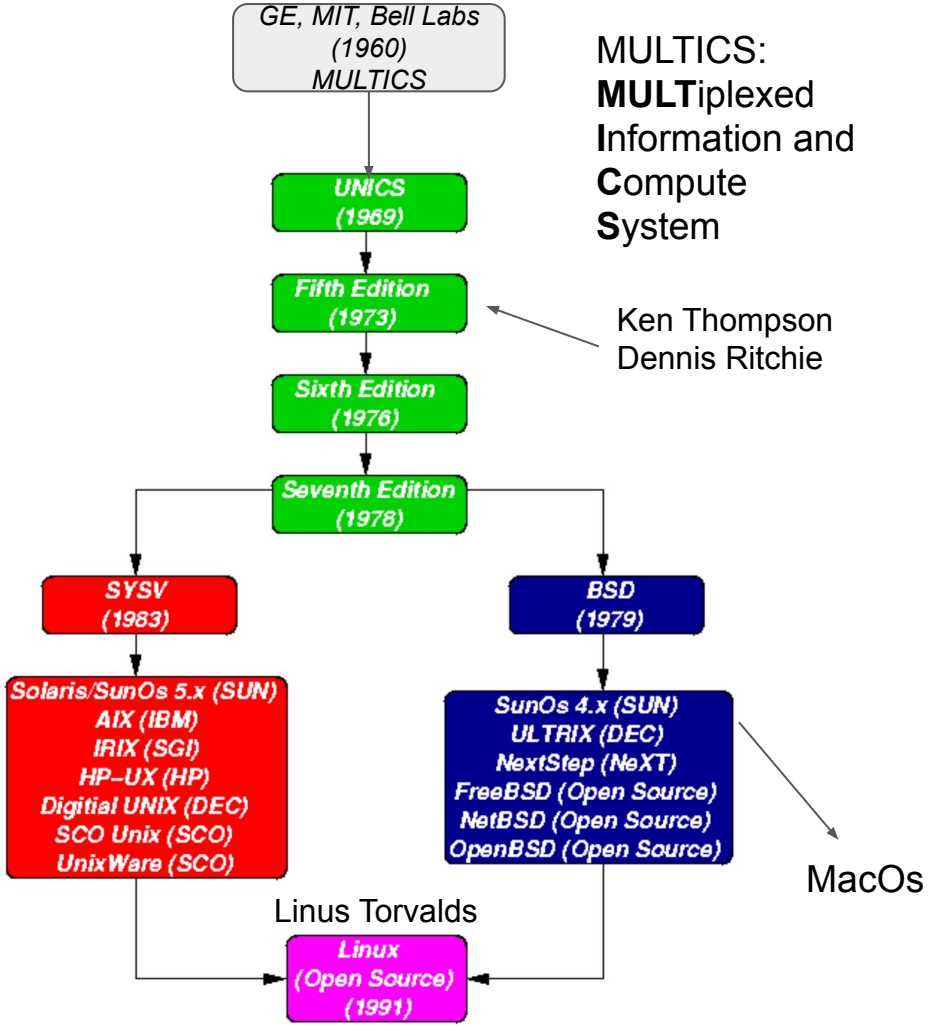
- A resource manager
- A set of software routines that allow users and application programs to access system resources (e.g. the CPU, memory) in a safe way.



source

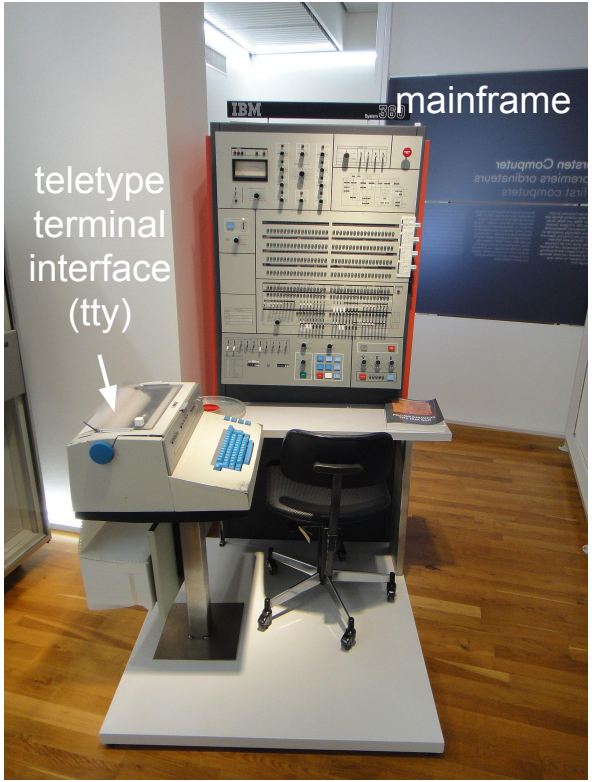
Brief History of Unix and Linux Operating Systems

- Unix and variants have been around for more than 20 years: multi-user, multi-tasking, networking, stability
- Support input through a command line shell (Terminal) as well as a GUI (Graphical User Interface)
- Examples of system utilities accessed through Terminal:
  - cp
  - ls
  - grep
  - mkdir



source

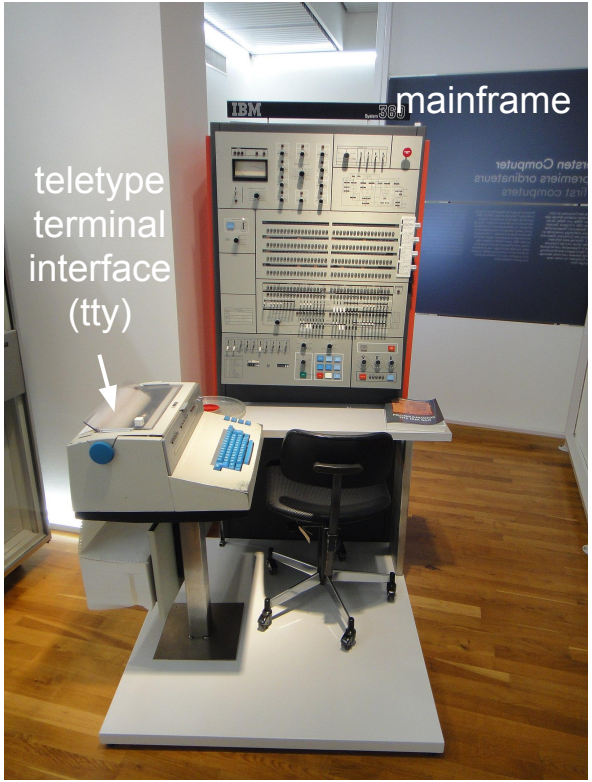
Working with Unix in the early days



By Sandstein - Own work, CC BY-SA 3.0,
https://commons.wikimedia.org/w/index.php?curid=16305441

By Gorthmog - Own work This file has been extracted from another file: DEC VT100 terminal.jpg, CC BY-SA 4.0,
https://commons.wikimedia.org/w/index.php?curid=78497051

Working with Unix in the early days



By Sandstein - Own work, CC BY-SA 3.0,
https://commons.wikimedia.org/w/index.php?curid=16305441

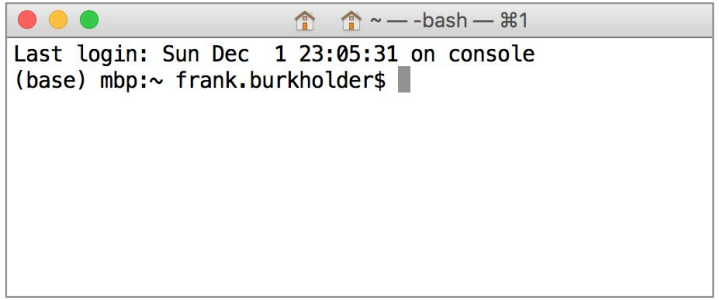
By Gorthmog - Own work This file has been extracted from
another file: DEC VT100 terminal.jpg, CC BY-SA 4.0,
https://commons.wikimedia.org/w/index.php?curid=78497051

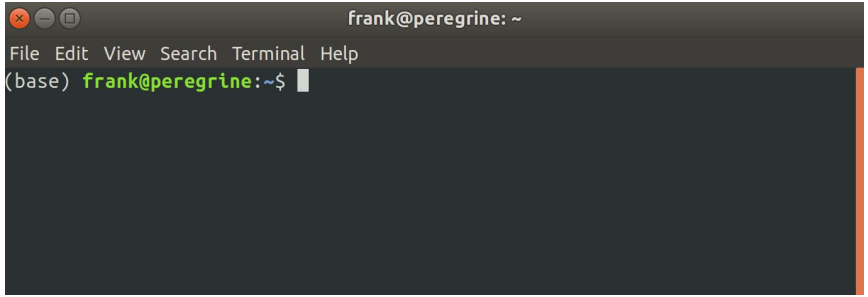
Terminal: text input/output environment

Console: the physical terminal

shell: the environment interpreting your commands

The (virtual) terminals in MacOs and Linux





MacOs:
command - space
(to open Spotlight)
then type
“terminal”

Ubuntu Linux:
ctrl - alt - t

Advantages to this command-line interface to this day

Unix -> Linux (open source) -> “secure”, free OS working on servers around the world

Linux is actively developed.

Unix (and Linux) have a sound design philosophy:

1. *Small is beautiful.*
2. *Make each program do one thing well.*
3. *Build a prototype as soon as possible.*
4. *Choose portability over efficiency.*
5. *Store data in flat text files.*
6. *Use software leverage to your advantage.*
7. *Use shell scripts to increase leverage and portability.*
8. *Avoid captive user interfaces.*
9. *Make every program a filter.*

Mike Gancarz, The Unix Philosophy, 1994

Source: Wikipedia - Unix Philosophy

Unix/Linux: easy data science model deployment

```
$ python main.py --mode train --data training.csv --model price_predictor.pkl
```

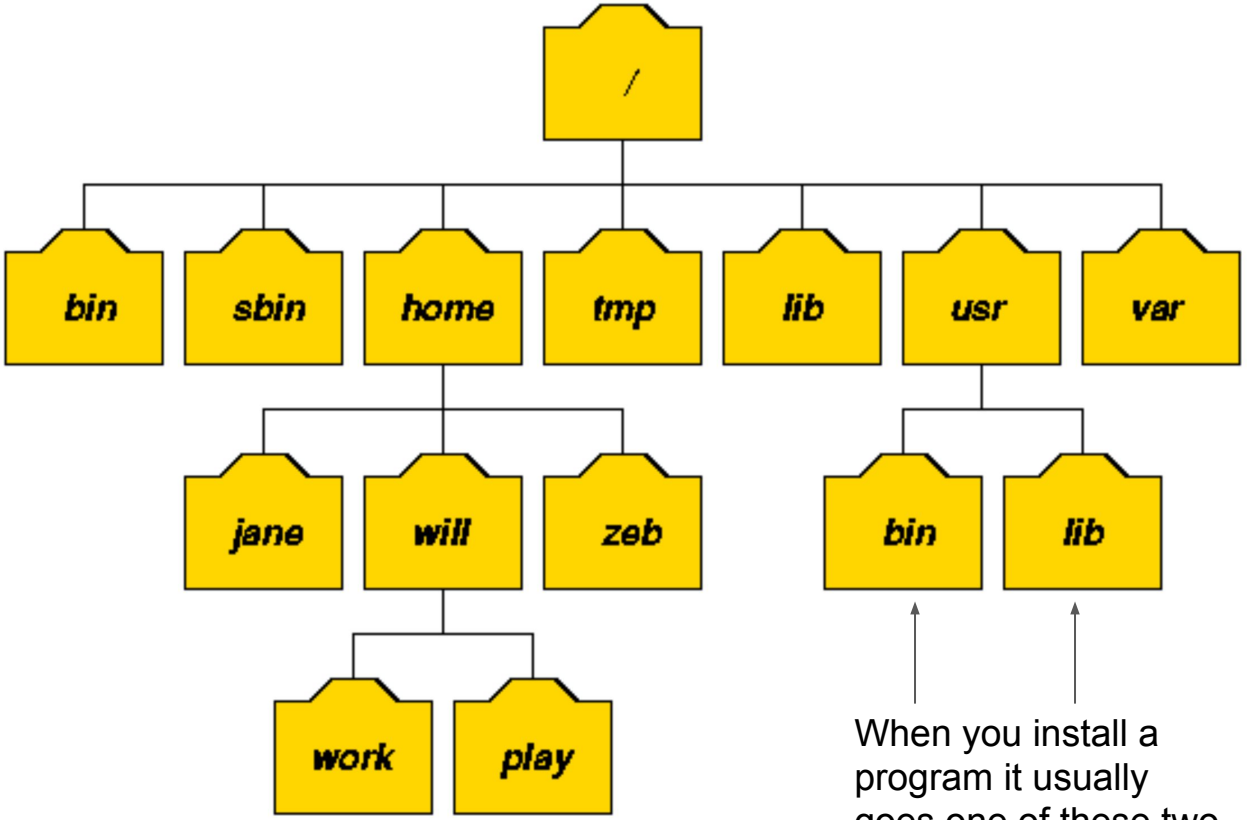
Python runs `main.py`, which uses command line argument parsing (Argparse) to take in arguments that affect how `main.py` executes. Here, `main.py` is running in `train` mode, so it will be expecting data with labels to train on in `training.csv`. After training, `main.py` will save the model and serialize it (pickle it), naming in `price_predictor.pkl`.

```
$ python main.py --mode predict --data june.csv --model price_predictor.pkl --out predictions.csv
```

Now `main.py` is running in `predict` mode, so it isn't expecting labels with the `june.csv` data. `main.py` is using the earlier trained model (`price_predictor.pkl`)to make the predictions, and saving those predictions in `predictions.csv`.

`predictions.csv` will likely be used by another program that will be executed after it in a shell script.

General Unix file structure



When you install a program it usually goes one of these two places

source

General Syntax of Unix commands

```
$ command -options targets ↵
```

And what are the options associated with that command?

```
$ man command
```

Breakout

- With a partner, use the man pages built in to Terminal, investigate `cp, ls, grep, mkdir, and less.` Specify
  - a) what the command does
  - b) two possible options you could give the command to modify/specify it's functionality.
- You have 5 minutes and then we'll ask groups to present.

Pipes

The pipe ('|') operator is used to create concurrently executing processes that pass data directly to one another. It is useful for combining system utilities to perform more complex functions. For example:

```
$ cat hello.txt | sort | uniq ↵
```

creates three processes (corresponding to `cat`, `sort` and `uniq`) which execute concurrently. As they execute, the output of the who process is passed on to the `sort` process which is in turn passed on to the `uniq` process. `uniq` displays its output on the screen (a sorted list of users with duplicate lines removed). Similarly:

```
$ cat hello.txt | grep "dog" | grep -v "cat" ↵
```

finds all lines in `hello.txt` that contain the string "`dog`" but do not contain the string "`cat`".

source

Redirecting Output

To redirect standard output to a file instead of the screen, we use the > operator:

```
$ echo hello ↵
hello
$ echo hello > output ↵
$ cat output ↵
hello
```

In this case, the contents of the file `output` will be destroyed if the file already exists. If instead we want to append the output of the echo command to the file, we can use the >> operator:

```
$ echo bye >> output ↵
$ cat output ↵
hello
bye
```

source

References

- Rob Funk - Introduction to Unix - University Technology Services - Ohio State

Objectives

- Define an Operating System
- History of Unix & Linux
- Advantages of the Unix philosophy and the command line
- General Unix file structure
- General syntax of Unix commands
- Overview of common commands
- Pipes
- Redirecting output

Appendix

Directory and File Handling Commands

- `pwd` (print [current] working directory)

  `pwd` displays the full absolute path to the your current location in the filesystem. So

  ```
  $ pwd ↵
  /usr/bin
  ```

  implies that `/usr/bin` is the current working directory.

- `ls` (list directory)

  `ls` lists the contents of a directory. If no target directory is given, then the contents of the current working directory are displayed. So, if the current working directory is `/`,

  ```
  $ ls ↵
  bin   dev  home  mnt   share  usr  var
  boot  etc  lib   proc  sbin   tmp  vol
  ```

  Actually, `ls` doesn't show you *all* the entries in a directory - files and directories that begin with a dot (.) are hidden (this includes the directories '.' and '..' which are always present). The reason for this is that files that begin with a . usually contain important configuration information and should not be changed under normal circumstances. If you want to see all files, `ls` supports the `-a` option:

source

Directory and File Handling Commands

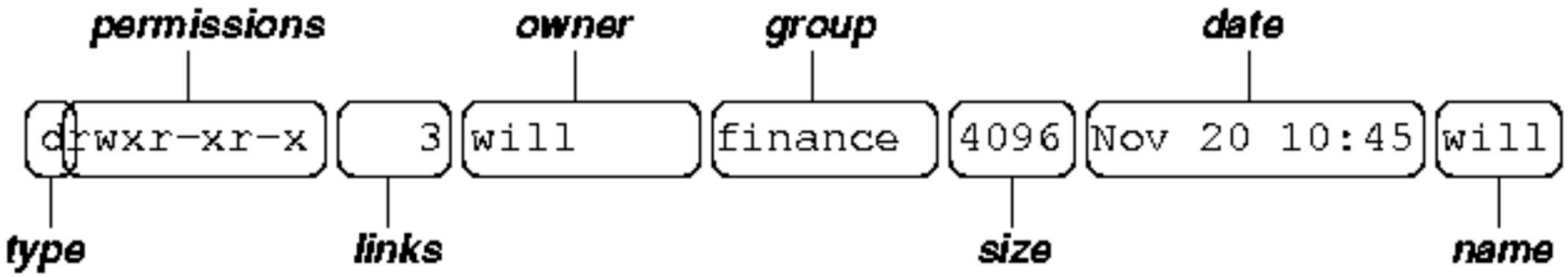
```
$ ls -a -l ↵
```

(or, equivalently,)

```
$ ls -al ↵
```

-a to see hidden files and directories
-l to see long listing

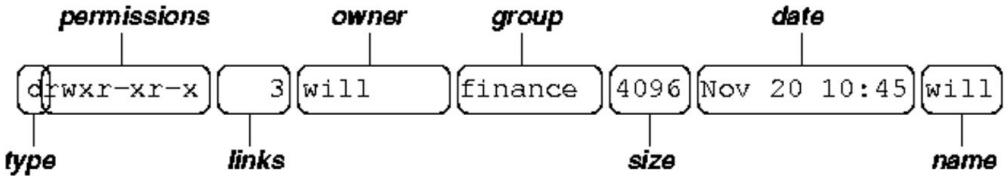
Each line of the output looks like this:



source

Directory and File Handling Commands

```
$ ls -al ↵
```



where:

- *type* is a single character which is either 'd' (directory), '-' (ordinary file), 'l' (symbolic link), 'b' (block-oriented device) or 'c' (character-oriented device).
- *permissions* is a set of characters describing access rights. There are 9 permission characters, describing 3 access types given to 3 user categories. The three access types are read ('r'), write ('w') and execute ('x'), and the three users categories are the user who owns the file, users in the group that the file belongs to and other users (the general public). An 'r', 'w' or 'x' character means the corresponding permission is present; a '-' means it is absent.
- *links* refers to the number of filesystem links pointing to the file/directory (see the discussion on hard/soft links in the next section).
- *owner* is usually the user who created the file or directory.
- *group* denotes a collection of users who are allowed to access the file according to the group access rights specified in the permissions field.
- *size* is the length of a file, or the number of bytes used by the operating system to store the list of files in a directory.
- *date* is the date when the file or directory was last modified (written to). The `-u` option display the time when the file was last accessed (read).
- *name* is the name of the file or directory.

source

Directory and File Handling Commands

- `cd` (change [current working] directory)

  `$ cd` *path*

  changes your current working directory to *path* (which can be an absolute or a relative path). One of the most common relative paths to use is '..' (i.e. the parent directory of the current directory).

  Used without any target directory

  `$ cd` ↵

  resets your current working directory to your home directory (useful if you get lost). If you change into a directory and you subsequently want to return to your original directory, use

  `$ cd -` ↵

source

Directory and File Handling Commands

- `mkdir` (make directory)

  `$ mkdir` *directory*

  creates a subdirectory called *directory* in the current working directory. You can only create subdirectories in a directory if you have write permission on that directory.

- `rmdir` (remove directory)

  `$ rmdir` *directory*

  removes the subdirectory *directory* from the current working directory. You can only remove subdirectories if they are completely empty (i.e. of all entries besides the '.' and '..' directories).

$ rm -r *directory*

deletes directory (no trash, be careful!)

Directory and File Handling Commands

- `cp` (copy)

  `cp` is used to make copies of files or entire directories. To copy files, use:

  `$ cp` *source-file(s) destination*

  where *source-file(s)* and *destination* specify the source and destination of the copy respectively. The behaviour of `cp` depends on whether the destination is a file or a directory. If the destination is a file, only one source file is allowed and `cp` makes a new file called *destination* that has the same contents as the source file. If the destination is a directory, many source files can be specified, each of which will be copied into the destination directory. Section 2.6 will discuss efficient specification of source files using wildcard characters.

  To copy entire directories (including their contents), use a *recursive* copy:

  `$ cp -rd` *source-directories destination-directory*

$ cp -r *directory_to_copy where_to_put_copy*

source

Directory and File Handling Commands

- `mv` (move/rename)

  `mv` is used to rename files/directories and/or move them from one directory into another. Exactly one source and one destination must be specified:

  $ mv *source destination*

  If *destination* is an existing directory, the new name for *source* (whether it be a file or a directory) will be *destination/source*. If *source* and *destination* are both files, *source* is renamed *destination*. N.B.: if *destination* is an existing file it will be destroyed and overwritten by *source* (you can use the `-i` option if you would like to be asked for confirmation before a file is overwritten in this way).

- `rm` (remove/delete)

  $ rm *target-file(s)*

  removes the specified files. Unlike other operating systems, it is almost impossible to recover a deleted file unless you have a backup (there is no recycle bin!) so use this command with care. If you would like to be asked before files are deleted, use the `-i` option:

```
$ rm -i myfile↵
rm: remove 'myfile'?
```

  `rm` can also be used to delete directories (along with all of their contents, including any subdirectories they contain). To do this, use the `-r` option. To avoid `rm` from asking any questions or giving errors (e.g. if the file doesn't exist) you used the `-f` (force) option. Extreme care needs to be taken when using this option - consider what would happen if a system administrator was trying to delete user `will`'s home directory and accidentally typed:

```
$ rm -rf / home/will↵
```

source

Directory and File Handling Commands

Displaying text files

- `more` and `less` (catenate with pause)

  `$ more` *target-file(s)*

  displays the contents of *target-file(s)* on the screen, pausing at the end of each screenful and asking the user to press a key (useful for long files). It also incorporates a searching facility (press '`/`' and then type a phrase that you want to look for).

  You can also use `more` to break up the output of commands that produce more than one screenful of output as follows (`|` is the pipe operator, which will be discussed in the next chapter):

  ```
  $ ls -l | more↵
  ```

  `less` is just like `more`, except that has a few extra features (such as allowing users to scroll backwards and forwards through the displayed file). `less` not a standard utility, however and may not be present on all UNIX systems.

File and Directory Permissions

Permission	File	Directory
read	User can look at the contents of the file	User can list the files in the directory
write	User can modify the contents of the file	User can create new files and remove existing files in the directory
execute	User can use the filename as a UNIX command	User can change into the directory, but cannot list the files unless (s)he has read permission. User can read files if (s)he has read permission on them.

File and directory permissions can only be modified by their owners, or by the superuser (`root`), by using the `chmod` system utility.

- `chmod` (change [file or directory] mode)

  `$ chmod` *options files*

  `chmod` accepts options in two forms. Firstly, permissions may be specified as a sequence of 3 octal digits (octal is like decimal except that the digit range is 0 to 7 instead of 0 to 9). Each octal digit represents the access permissions for the user/owner, group and others respectively. The mappings of permissions onto their corresponding octal digits is as follows:

---	**0**
--x	**1**
-w-	**2**
-wx	**3**
r--	**4**
r-x	**5**
rw-	**6**
rwx	**7**

  For example the command:

  ```
  $ chmod 600 private.txt
  ```

  sets the permissions on `private.txt` to `rw-------` (i.e. only the owner can read and write to the file).

source

File and Directory Permissions

Permission	File	Directory
read	User can look at the contents of the file	User can list the files in the directory
write	User can modify the contents of the file	User can create new files and remove existing files in the directory
execute	User can use the filename as a UNIX command	User can change into the directory, but cannot list the files unless (s)he has read permission. User can read files if (s)he has read permission on them.

File and directory permissions can only be modified by their owners, or by the superuser (`root`), by using the `chmod` system utility.

- `chmod` (change [file or directory] mode)

  `$ chmod` *options files*

  `chmod` accepts options in two forms. Firstly, permissions may be specified as a sequence of 3 octal digits (octal is like decimal except that the digit range is 0 to 7 instead of 0 to 9). Each octal digit represents the access permissions for the user/owner, group and others respectively. The mappings of permissions onto their corresponding octal digits is as follows:

---	0
--x	1
-w-	2
-wx	3
r--	4
r-x	5
rw-	6
rwx	7

Let's say you want to run a file, but you are told you don't have permission. What do you have to change?

When you try to see the files in a given directory, you are told you don't have permission. What would you change?

For example the command:

```
$ chmod 600 private.txt
```

sets the permissions on `private.txt` to `rw-------` (i.e. only the owner can read and write to the file).

source

Finding Text of Interest in Text Files

- `grep` (General Regular Expression Print)

  `$ grep` *options pattern files* ↵

  `grep` searches the named files (or standard input if no files are named) for lines that match a given pattern. The default behaviour of `grep` is to print out the matching lines. For example:

  `$ grep hello *.txt` ↵

  searches all text files in the current directory for lines containing "hello". Some of the more useful options that `grep` provides are:
  `-c` (print a count of the number of lines that match), `-i` (ignore case), `-v` (print out the lines that don't match the pattern) and `-n` (printout the line number before printing the matching line). So

  `$ grep -vi hello *.txt` ↵

  searches all text files in the current directory for lines that do not contain any form of the word hello (e.g. Hello, HELLO, or hELlO).

  If you want to search all files in an entire directory tree for a particular pattern, you can combine `grep` with `find` using backward single quotes to pass the output from `find` into `grep`. So

  `$ grep hello` ` `find . -name "*.txt" -print` ` ↵

source

Shell and Shell Scripts

A shell lets you define variables (like most programming languages). A variable is a piece of data that is given a name. Once you have assigned a value to a variable, you access its value by prepending a `$` to the name:

```
$ bob='hello world' ↵
$ echo $bob
hello world
$
```

Variables created within a shell are local to that shell, so only that shell can access them. The `set` command will show you a list of all variables currently defined in a shell. If you wish a variable to be accessible to commands outside the shell, you can *export* it into the *environment*:

```
$ export bob ↵
```

Shell and Shell Scripts

To execute this script, we first have to make the file `simple` executable:

```
$ ls -l simple↵
-rw-r--r--    1 will  finance  175  Dec 13  simple
$ chmod +x simple↵
$ ls -l simple↵
-rwxr-xr-x    1 will  finance  175  Dec 13  simple
$ ./simple hello world↵
The number of arguments is 2
The arguments are hello world
The first is hello
My process number is 2669
Enter a number from the keyboard:
5↵
The number you entered was 5
$
```

source