ENGG1340 Computer Programming II
COMP2113 Programming Technologies
Module 1: Linux Environment
Estimated time to complete: 4 hours

Objectives

At the end of this chapter, you should be able to:

- Start working in the Linux environment.
- Start using the Bash shell.
- Display and understand the manuals of some simple shell commands.

Table of Contents

1. Introduction to Linux

1.1 What is Linux?

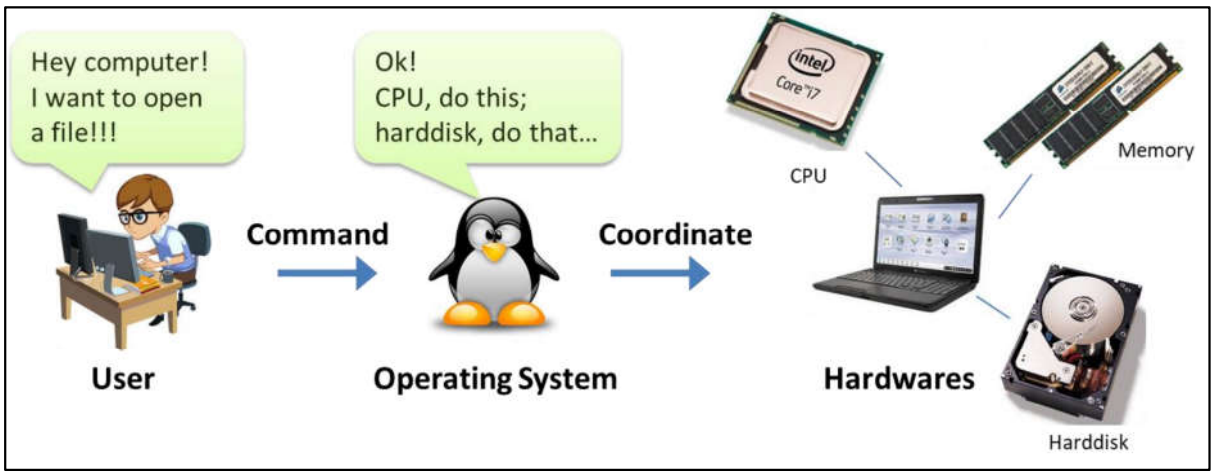


Figure 1 The Linux operating system provides an interface for users to access the hardware.

Linux is an **operating system** like Microsoft Windows or Apple macOS (formerly known as OSX). Imagine we have connected the hardware components of a computer, e.g., the CPU, hard disks, the keyboard, etc. How can we coordinate the operations of these components to perform the tasks like copying a file? This is the role of an operating system.

It provides an interface for us to use the hardware components. Most operating systems also come with tools for many common tasks, like text editing or web browsing.

Figure 2 Popular distributions of Linux OSs

In this course, we are using **Ubuntu**. All contents of this course were prepared based on Ubuntu on the CS academy server. However, there are many different distributions of Linux. They deviate from each other mainly on the interfaces and tools provided.

You are welcome to use other distributions like Fedora or Debian (http://www.debian.org/). For this course, these distributions may not make a difference.

1.1.1 Why do I have to learn Linux?

Windows and macOS (OSX) are probably more commonly known than Linux by the public. However, Linux is an important operating system for some reasons.

1. Linux is **open-source** software. Hence, we can customize it for new or special hardware systems. Hence, Linux is particularly popular for supercomputers. It also provides a higher level of security because a company can inspect its source code.

2. A lot of software development tools are available on Linux, e.g., C, C++, python, debugging, timing, and profiling tools. It largely eases software development.

3. Linux and most software provided for Linux can be used **without paying a fee**. Hence, it can be used to produce low price products like netbooks and mobile phones. For example, the Android operating system is developed based on Linux.

1.2 Using Linux

Besides installing a Linux OS on your computer, you can remote access the Ubuntu Linux server in the CS department via X2Go client (with GUI) or SSH (without GUI). For more details on using X2Go and SSH to access the CS server, please refer to Module 0.

1.3 Using Linux Shell

We can conveniently use Linux by entering commands into a shell. A shell is a text-based program that accepts user commands and performs the corresponding tasks.

- There are some different shells.
  - Korn Shell
  - Bourne Shell
  - C Shell
  - Bash Shell
- Note that commands are not unified among different types of shell, e.g., the Korn shell uses "`print`" to print out a string, while the Bash shell uses "`echo`".
- The default shell in Fedora is the Bash Shell. We will focus on Bash Shell in this course.

1.3.1 Start a Shell

- If you are using X2Go to connect the server, launch a bash Shell by
  Clicking "Applications" > "Terminal Emulator"

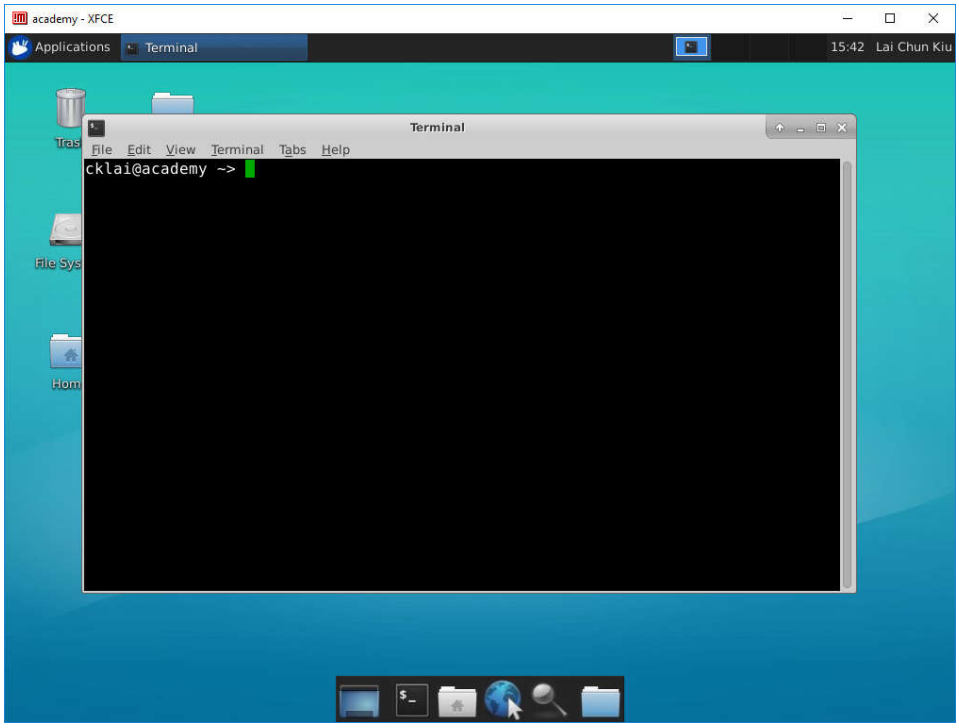


Figure 3 Terminal is your Shell

- If you are using SSH, the command-line interface is the Shell.

The shell accepts commands and performs specific tasks. For example, to check current datetime, you need to type the **`date`** command as follow:

```
$ date
Fri Jul 27 15:58:42 HKT 2018
```

*Note that the $ sign indicates the command line in the shell; thus, you do not need to type it.

1.3.2 Listing Directories and Files

All data in Linux is organized into files, and all files are organized into directories (known as folders). The filesystem in Linux is a tree structure organized by these directories.

The **ls** command lists all the files and directories in your current directory.

```
$ ls
Documents/  log.bat*  public_html/  test.txt  Downloads/
Pictures/  Templates/    Videos/    Desktop/  engg1340/
Music/      Public/    test2.txt
```

The **cd** command is to navigate your current directory to another. E.g., To access the directory "engg1340" and then list out all the files and directories.

```
$ cd engg1340
$ ls
lab1/  lab2/  lab3/
```

To navigate to the parent directory, type "**cd ..**".

The **pwd** command is to print the current working directory. This is a useful command to know where you are in the Linux filesystem hierarchy.

```
$ pwd
$ /home/research/ra/1801/tmchan
```

1.3.3 Display the manual of a command

The shell supports many commands. Therefore, it is impossible to remember all the Linux commands. Here is the command that helps you to get the detail of a command – **man**

The "**man x**" command returns the manual page of the command x.

Suppose we want to use the **man** command to know more about the **ls** command.

```
$ man ls
```

```
LS(1)                          User Commands                          LS(1)
NAME
       ls - list directory contents
SYNOPSIS
       ls [OPTION]... [FILE]...
DESCRIPTION
       List  information  about  the FILEs (the current directory by default).
       Sort entries alphabetically if none of -cftuvSUX nor --sort  is  speci-
       fied.
       Mandatory  arguments  to  long  options are mandatory for short options
       too.

       -a, --all
              do not ignore entries starting with .
       -A, --almost-all
              do not list implied . and ..
       --author
 Manual page ls(1) line 1/234 8% (press h for help or q to quit)
```

*Press arrow keys (↑, ↓) to scroll the manual page.
*Press 'Q' to close the manual page.

From the NAME section, we learn that the function of the **ls** command is to "list directory contents".

From the DESCRIPTION section, we learn that "**-l**" option is to "use a long listing format". Long listing means that we also list the file size, owners, last modification date, etc. We will learn more about the long list format in the next section of this document.

Try the command **ls**; it will display the current directory content.

```
$ ls
```

How about displaying the "long listing format" of the current directory content?

```
$ ls -l
```

2. Directory and File Management

2.1 Linux Directory

The file structure of Linux starts with the **root** directory, denoted as **"/"**, which contains all other files and directories.

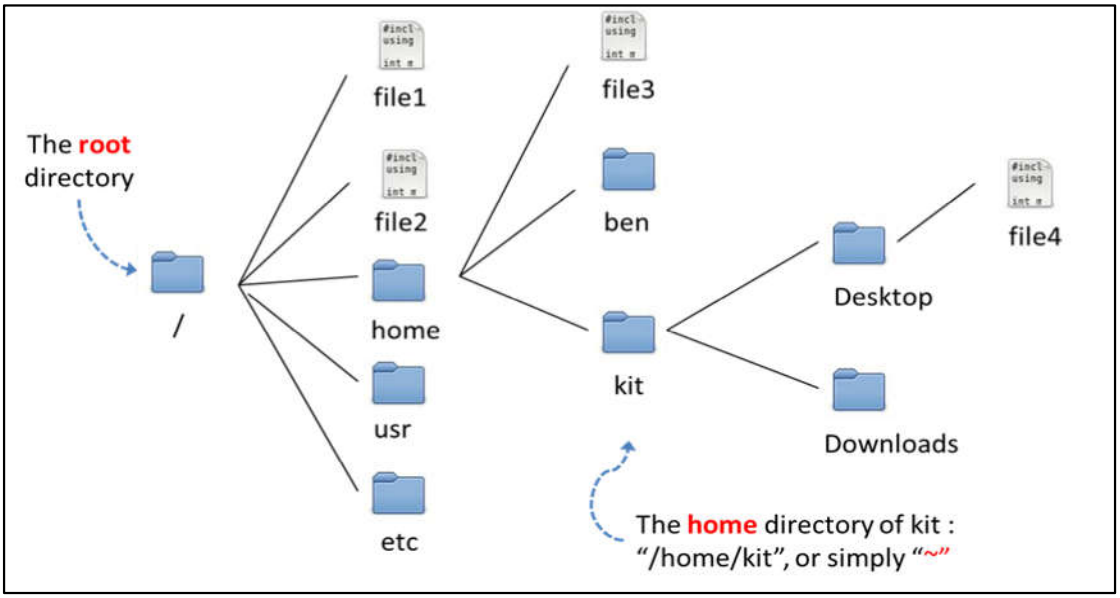


Figure 4 A common file structure in a Linux computer

All files and directories can be specified using a unique path. There are two options: full path and relative path:

Full path always starts with a **"/"** and concatenates all directory names from the root to the specific files. As an example, in Figure 4:

- **/home/file3** is the full path for **file3** under the **/home** directory.
- **/home/kit/Desktop/file4** is the full path for **file4**.

Relative path is relative to the present working directory.

- A relative path does not start with a **"/"**.
- Consider Figure 4 above, if the present working directory is **/home**, then the following path would refer to the **file3** under **/home**.

  ```
  file3
  ```

  *Note: this path does not start with a **"/"**, so it is a relative path.

- If the present working directory is **/home/kit**, we can refer to the **file4** under **Desktop** using the following relative path:

  ```
  Desktop/file4
  ```

If a directory **A** contains another directory **B**, then **A** is called the parent directory of **B**, and **B** is called a subdirectory of **A**. For example, **kit** is a subdirectory of **home** and **home** is the parent directory of **kit**.

Once the shell starts, its present working directory is the **home** directory. We can always get the **present working directory** by the command **`pwd`**.

2.1.1 Home Directory

For any user, there is a home directory for that user. For example, the user kit will have a home directory at /home/kit.

Here "~" denote the home directory of a user. You can go to your **home** directory using:

```
$ cd ~
```

You can go to any other user's home directory:

```
$ cd ~username
```

2.2 Directory Management

2.2.1 Create Directories

The **`mkdir`** command is to create directories.

E.g., Create a directory *lab* in the current directory:

```
$ mkdir lab
```

If you want to create more than one directory, give them on the command line:

```
$ mkdir lab1 lab2
```

Naming a directory with space characters can be done using the double quotation mark (**`""`**).

```
$ mkdir "lab 1"
```

2.2.2 Remove Directories

An empty directory can be deleted using the **`rmdir`** command:

```
$ rmdir lab
```

For a non-empty directory, we can use the **`rm -rf`** or **`rm -r -f`** command to delete the entire directory

```
$ ls lab1
file1.txt  file2.txt
$ rm -r -f lab1
$ ls lab1
ls: cannot access 'lab1': No such file or directory
```

- The **`-r`** flag recursively travels down any subdirectory and removes all files and directories.
- The **`-f`** makes sure you are not prompted for confirmation (handy if there are lots of files and directories inside the target directory)

2.2.3 Rename Directories

The **mv** (move) command can rename a directory. For example, rename directory *lab* to *lab 1*:

```
$ mv lab "lab 1"
```

*The 1st and 2nd arguments are directories

2.2.4 Useful Commands

Below shows a list of useful commands for directory manipulation. Some commands can be invoked with different flags or arguments. We list those variations in the table as well.

Command	Meaning
`pwd`	It prints the name of the present working directory.
`ls` • `ls -l` • `ls -a` • `ls -la`	It lists the content in the present working directory. – It lists the content in long format, which contains the file size, owners, last modification date, etc. – It lists all content, including hidden files or hidden directories. Notice that a file or directory is hidden if its name starts with a dot. – It has both the effect of `-l` and `-a`
`cd dir` • `cd ~` • `cd ..` • `cd .`	It changes the current directory to dir. – Changes to the home directory. – Changes to the parent directory. – Changes to the current directory. Hence, this command is valid yet has no effect actually.
`mkdir dir`	It creates a new directory with the name *dir*.
`rmdir dir`	It removes the directory *dir*. This only works if *dir* is empty.
`rm -rf dir`	It removes the non-empty directory *dir* and all the subdirectories / files.
`mv dir dir2`	It renames the directory from *dir* to *dir2*.

Note that "**-l**" and "**-a**" are called **flags** to the command, which changes the behavior of the command. On the other hand, *dir* is called an **argument** of the command, which specifies the target of the command.

2.3 File Management

2.3.1 Create files

Use the touch command to create an empty file called *file1.txt*

```
$ touch file1.txt
```

2.3.2 Display files

Use the **cat** command to display the content of a file.

```
$ cat file1.txt
Hello, this is the content of file1.txt
Bye Bye!
```

2.3.3 Move files

The **mv** command can move a file to a directory.

E.g., move *hello.txt* to *mydir* directory.

```
$ mv hello.txt mydir
```

*The 1st argument is a file, while the 2nd is a directory.

If the files have the same prefixes, you can specify all of them at once using the asterisk symbol (*).

Move all files matching with the prefix "**myfile**" to the *lab* directory.

```
$ ls
myfile1.txt  myfile2.txt  lab/
```

```
$ mv myfile* lab
```

```
$ ls
lab/
```

```
$ ls lab
myfile1.txt  myfile2.txt
```

2.3.4 Useful Commands

Below shows a list of useful commands for file management. Some commands can be invoked with different flags or arguments. We list those variations in the table as well.

Command	Meaning
`cp file1 file2` `cp -r dir1 dir2`	Copy *file1* into *file2*. Copy *dir1* into *dir2* including sub-directories

`mv file dir`
`mv file1 file2`
`mv dir1 dir2`
	If *dir* is a directory, it **moves** the *file* into *dir*.
	If the two arguments are the same type (e.g., both *file1* and *file2* are files, it **renames** *file1* to *file2*. (The same for directories)

`rm file` `rm -rf dir`	Remove *file*. Remove all files and directories recursively in *dir*
`touch file`	Create an empty file named *file*.
`cat file`	Display the content of a *file*.

2.3.5 Use of vi editor

In this section, we will discuss the use of the vi editor for creating and editing files. vi is a command-line text editor (like Notepad on Windows or gedit on Linux). However, it does not come with a GUI (Graphical User-interface).

There are two modes of operation in the vi editor.

- Insert mode – you may edit text on the file.
- Command mode – perform commands such as saving a file.

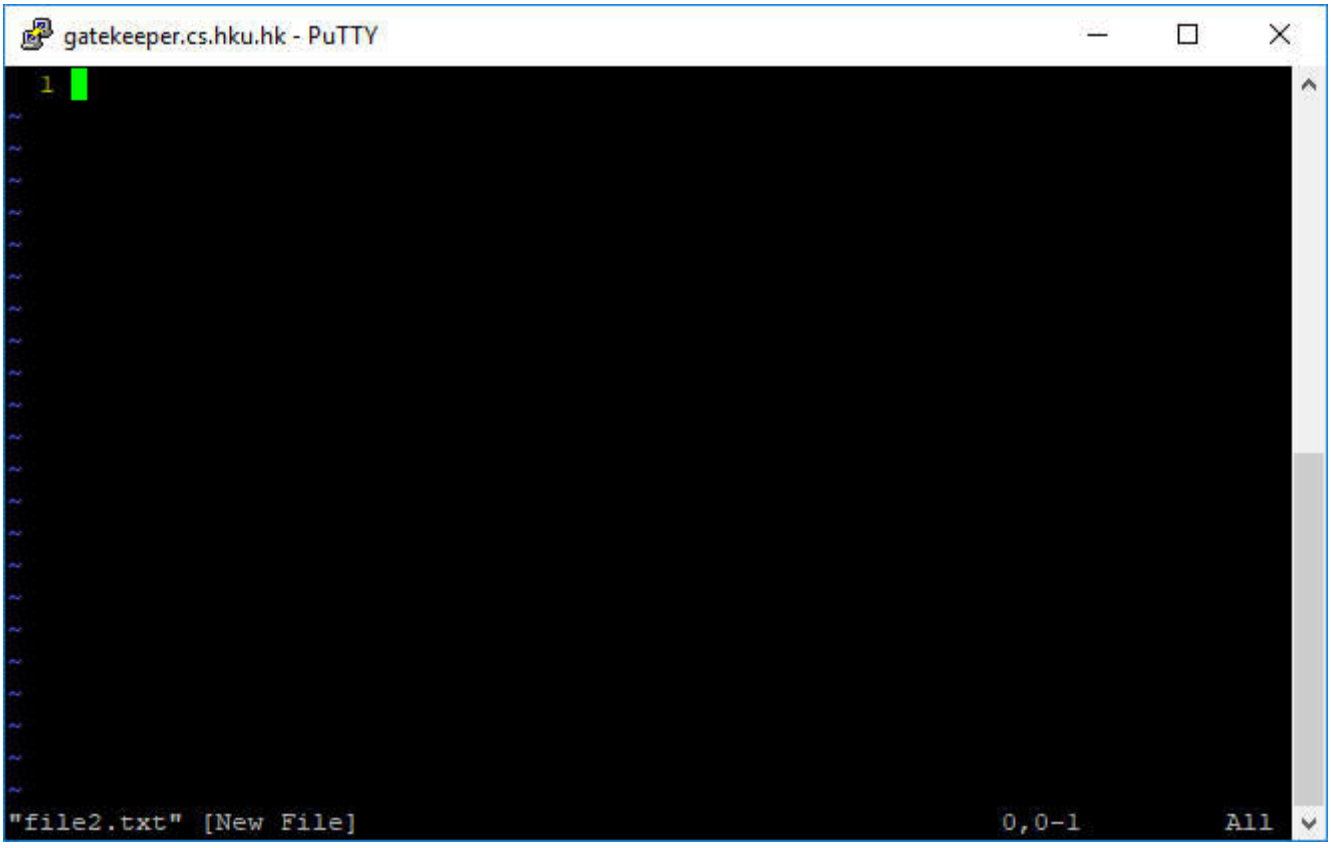
Press the 'Esc' key to enter **Command mode**. Press 'I' to enter **Insert mode**.

Use vi on a file

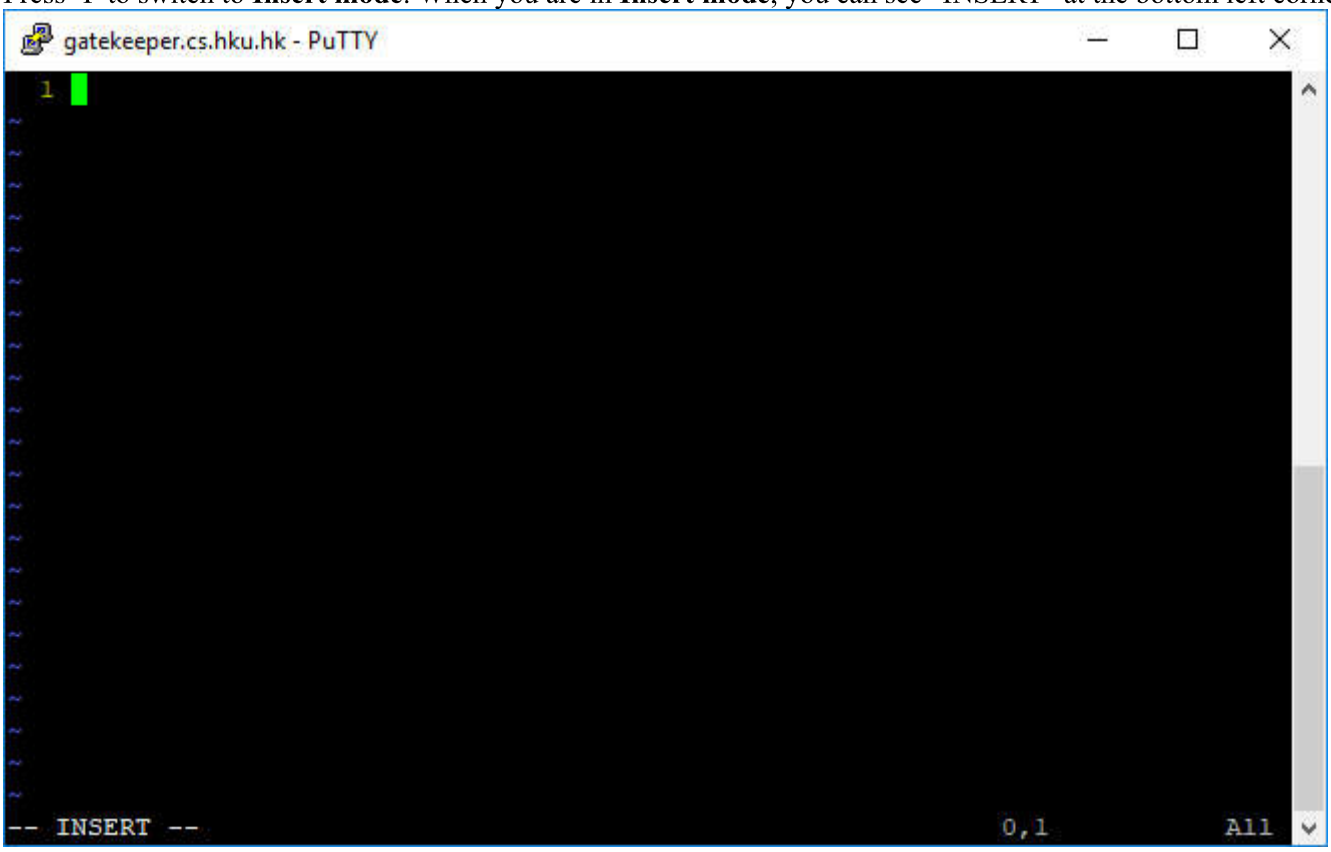
Type **vi *filename*** to open a file. If the file does not exist, the editor will create a new one for you.

```
$ vi file2.txt
```

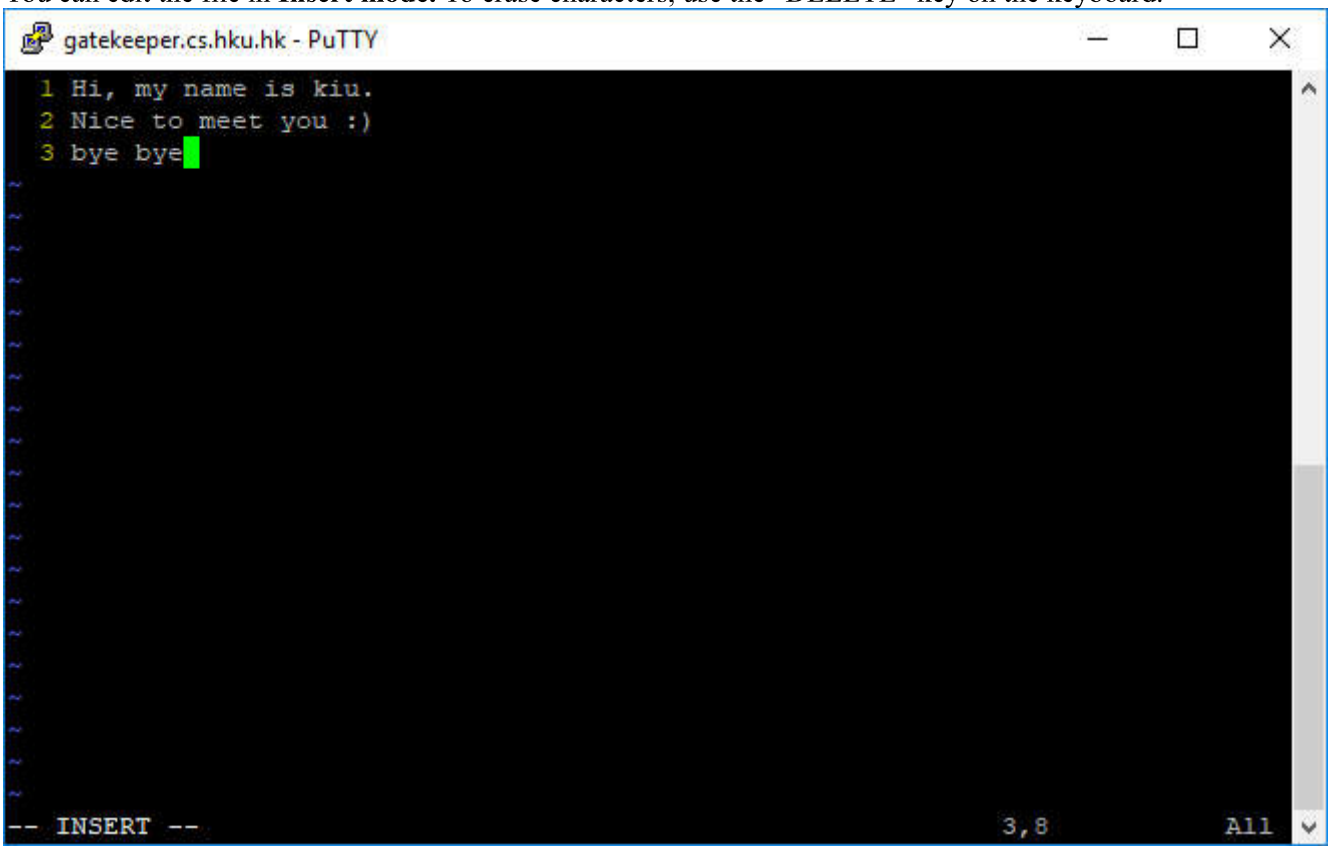
If the file exists, the content of the file will be displayed. Here, *files2.txt* does not exist, so an empty file is created and displayed. When the vi editor is first launched, you are in **Command mode**.



Press ‘I’ to switch to **Insert mode**. When you are in **Insert mode**, you can see “INSERT” at the bottom left corner.



You can edit the file in **Insert mode.** To erase characters, use the “DELETE” key on the keyboard.

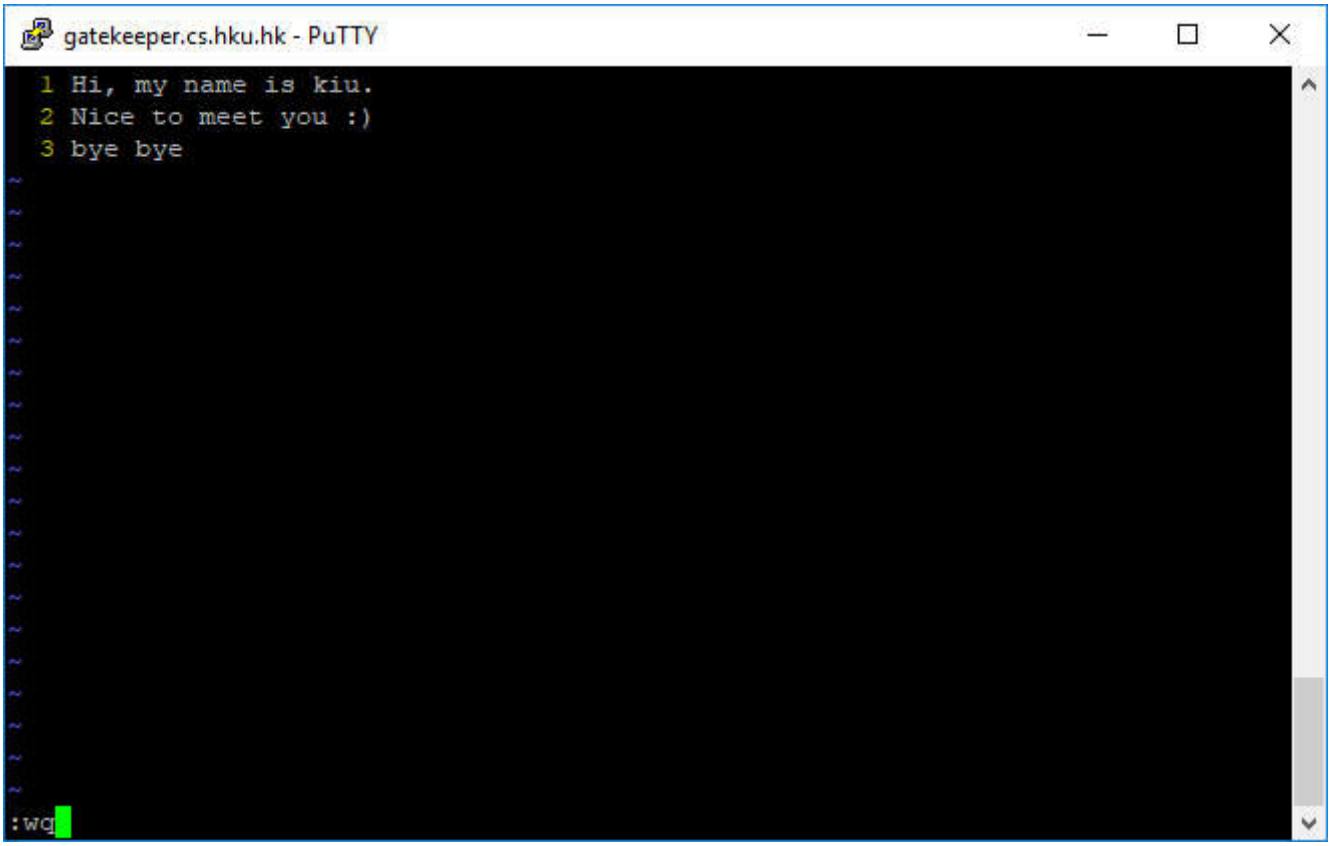


Save file and Quit

After editing, you need to enter **Command mode** to save the file.

Press 'ESC' to enter **Command mode**. In command mode, you don't see 'INSERT' in the bottom left corner.

Then enter the command "`:wq`", you can see the cursor will move to the bottom.



After pressing 'ENTER', the files will be saved, and you will exit from vi.

Commands

Other than the "save and exit" (`:wq`) command, you can save the file only without terminating the vi editor by using the ":w" command. To continue editing, press "ESC" and then press 'I' to switch back to **Insert mode**.

Below are some of the useful commands you can use in the command mode.

`:wq`	Save and Quit
`:w`	Save
`:w` *`filename`*	Save to new file named *filename*
`:q`	Quit
`:q!`	Quit without saving the file

For the full list of commands and references, please go to:

https://www.cs.colostate.edu/helpdocs/vi.html

or refer vi cheat sheet (See appendix 10.1)

2.3.6 Transfer Files to Linux Server

Sometimes, you may need to transfer files to your Linux server from external sources. For example, you may want to copy and paste codes from this note or download multiple files from Moodle/your host computer to the academy server. There are many ways to do it. Below methods are some of the ways.

Method 1: Use X2Go client

After login to the server via the X2Go client, you may use a web browser to download files from the web.

Method 2: Use Vi editor

For this method, please refer to the previous section on how to use the vi editor to create files.

Method 3: Use the cat command

For example, you can create a new file with the name *hello.txt* using the **`cat`** command and then type in/paste the content.

```
$ cat > hello.txt
Hi! I am using cat command to create a file.
I am using shell like a pro!
```

When you finish typing, press ‘Ctrl + D’ to save the file. Then, press ‘Ctrl + C’ to exit from the cat command.

Method 4: Use SFTP (Recommended)

The SFTP (SSH File Transfer Protocol) allows you to transfer files between the host computer and the remote server.

2.4 File Permission

In this section, we will discuss file permission in Linux.

Each file or directory in Linux System is assigned three types of owner.

- **User**: A user is the owner of the file.
- **Group**: A group can have multiple users. All users belonging to a group have the same access permissions to the file.
- **Other**: Any other user who has access to a file. The person has neither created the file nor belongs to a group that owns the file.

Each file or directory has three permission defined for User, Group, and Other.

- **Read**: Give you the permission to open and read a file. Read permission on a directory gives you the ability to list its content.
- **Write**: Give you the permission to modify the content of a file. Write permission on a directory gives you the ability to add, remove and rename files.
- **Execute**: Give you the permission to run a program.

Use **`ls -l`** command to list the directory. It provides different information about file permission.

```
$ ls -l
```

```
total 10
-rw------- 1 cklai ra 50 Jul 30 17:07 file1.txt
-rw------- 1 cklai ra 48 Jul 31 15:14 file2.txt
drwx------ 2 cklai ra  2 Jul 30 17:00 lab/
```

2.4.1 Permission Indicator

If we go through the output of the first two lines, we can break down the information **`ls -l`** gives us.

Let's look at the first 2 two rows of the result.

```
total 10
-rw------- 1 cklai ra 50 Jul 30 17:07 file1.txt
```

Explanations

Field	Description
`total 10`	How much space the files have taken up in the directory, in kilobytes.
`-rw-------`	**Important!!** This is the permission indicator of the file; we will discuss the meaning of it in more detail later.
`1`	This is the number of hard links the file has. (Hard links will not be discussed in this course)
`cklai`	The owner of the file.
`ra`	The default group that **cklai** belongs to (The group is called **ra**).
`50`	This is the file size in bytes.
`Jul 23 17:07`	Last modification time.
`file1.txt`	The file name.

What do the permission indicators represent?

The field of permission indicators is 10-character long. E.g. (`-rw-------`)

It is divided into four parts:

Type	User permission			Group Permission			Other permission		
[- / d]	r	w	x	r	w	x	r	w	x

Parts	Explanations
Type	If it is a dash "`-`", that means it is a normal file. If it is a "`d`", it means it is a directory. There are other possible values in this field; please refer to the references for more information.
User permissions	The next 3 bits represent the permission for the owner of the file. If the permission is "`rw-`", the owner can: • Read the file (`r`), • Write into the file (`w`) • But not execute the file (this file is just a text file, not supposed to be executed). If the file can be executed, then the dash will be replaced by the letter "`x`".
Group permission	The next three bits are the permission for the group you belong to. A group can consist of other members.
Other permission	The last three bits (`---`) are the permissions for anybody else. It is now "`---`", which means everybody cannot read the file, write content into the file (no `w`), nor execute the file (no `x`).

2.4.2 Change Permissions

You can change the permission of the files or directories you own.

Use the command **`chmod`** to change the permission. It has the following format:

```
chmod [who][operator][permissions] filename
```

Field *`who`* indicates whether we are altering the permission for user, group, other, or all user.

Values	Meaning
u	The user (owner) permissions
g	The group permissions
o	The other permissions
a	ALL (including user, group, and other)

The field *operator* indicates the action to be done

Values	Meaning
+	Add a permission
-	Take away a permission
=	Set the permission

Field *permission* indicates what permission to be granted or removed

Values	Meaning
r	Read permission
w	Write permission
x	Execute permission

Example

Use **touch** to create a new file, the default permission for a user is "read and write only" as the permission indicators for the user are "rw-". Use **chmod** to grant the write permission to others so that everyone can modify the *file*.

```
$ touch file
```

```
$ ls -l
total 26
-rw------- 1 cklai ra   0 Aug  1 11:48 file
```

```
$ chmod o+w file
```

```
$ ls -l
total 26
-rw-----w- 1 cklai ra   0 Aug  1 11:48 file
```

Where "**chmod o+w file**" means to add (**+**) the write (**w**) permission to other (**o**).

The default permission depends on the system and how you access (e.g., bash via SSH or Terminal in X2Go) it. The default setting may vary.

Change multiple permission at one time

We can also grant/remove multiple permissions in one command.

For example, grant the Read (r) and Execute(x) permissions for the other (o).

```
$ ls -l
-rw-----w- 1 cklai ra   0 Aug  1 11:48 file
```

```
$ chmod o+rx file
```

```
$ ls -l
-rw----rwx 1 cklai ra   0 Aug  1 11:48 file*
```

If a file is marked with (*) at the end of its name, it means the *file* is executable.

3. Other Useful Commands

Command	Description
`grep 'abc' file`	It **returns the lines** in the file that contains `"abc"`. More sophisticated pattern matching can be specified using the regular expression.
`cut -d, -f2 file`	It returns specific **columns of data**. It divides each line according to the delimiter specified by the flag **-d** and returns the column specified by the flag **-f** (the field number starts from 1).
`diff file1 file2`	Display lines that are different in *file1* and *file2*. Intuitively, **`diff`** matches all lines that are common in both files and displays those unmatched lines.
`wc file`	It counts the number of lines, words, and characters in the file.
`sort file`	It sorts the lines of the file in alphabetical order.
`uniq file`	It removes adjacent duplicate lines so that only one of the duplicated lines remains.
`spell file`	It displays all incorrect words in the file.
`su`	Changes the user into the superuser mode.

`yum install prog`
or
`apt-get install prog`
	It connects to the internet database, downloads, and installs the program prog.
	Usually, you need to run the **`su`** command to obtain the superuser right before you can install the program.

`man cmd`	Shows the manual page for the command **`cmd`**. It is very useful for you to find other options for using a command.

3.1 Search in a file (grep)

Use grep to search lines in a file

For example, to search lines that contain 'ke'.

```
$ grep 'ke' example1.txt
4 Chicken 50
1 Coke 5.5
```

* We will discuss grep command in detail in the later section.

```
5 Apple 3.5
4 Chicken 50
1 Coke 5.5
10 Jelly 5
3 Chocolate 15
2 Milk 8
```

example1.txt

3.2 Word Count (wc)

We can use the **`wc`** command to get the word count information of a file

```
$ wc example1.txt
 6 18 71 example1.txt
```

- 6 is the number of lines
- 18 is the number of words
- 71 is the number of bytes

Use the flag **`-l`** to return the number of lines only.

```
$ wc -l example1.txt
6 example1.txt
```

Use the flag **-w** to return the number of words only.

```
$ wc -w example1.txt
18 example1.txt
```

3.3 Sorting

Use the **sort** command to sort a file. Without any options, it sort in **alphabetical order**.

```
$ sort example1.txt
1 Coke 5.5
10 Jelly 5
2 Milk 8
3 Chocolate 15
4 Chicken 50
5 Apple 3.5
```

Note: The first column in the result is "1 Coke 5.5" but not "10 Jelly 5". It is because the sorting is in alphabetical order by default. However, the implementation may vary in different Linux systems, resulting in "10 Jelly 5" placed before "1 Coke 5.5".

Use the flag **-n** for **numeric sort**

```
$ sort -n example1.txt
1 Coke 5.5
2 Milk 8
3 Chocolate 15
4 Chicken 50
5 Apple 3.5
10 Jelly 5
```

Use the flag **-r** for sorting in **reversing order**

```
$ sort -n -r example1.txt
10 Jelly 5
5 Apple 3.5
4 Chicken 50
3 Chocolate 15
2 Milk 8
1 Coke 5.5
```

Use the flag **-k** for sorting on a **specific sort key field**. (Note that the field ID starts from 1 but not 0)

For example, to sort the data according to the **3rd field**.

```
$ sort -k3 -n example1.txt
5 Apple 3.5
10 Jelly 5
```

```
1 Coke 5.5
2 Milk 8
3 Chocolate 15
4 Chicken 50
```

Use the flag **`-t`** for specifying the **field separator** if the fields are not separated by space (space is the default delimiter for sort).

For example, if the fields are separated by commas (,), and we want to sort by the 3rd column, we need to specify the field separator "**`-t,`**".

```
$ cat example1_comma.txt
5,Apple,3.5
4,Chicken,50
1,Coke,5.5
10,Jelly,5
3,Chocolate,15
2,Milk,8
```

```
$ sort -t, -k3 -n example1_comma.txt
5,Apple,3.5
10,Jelly,5
1,Coke,5.5
2,Milk,8
3,Chocolate,15
4,Chicken,50
```

3.4 Cutting

The command **`cut`** returns the specific **column of data**.

We need to specify the delimiter (even it is a space), the flag for specifying delimiter is **`-d`**. (Do not mix up with the field separator of the command sort).

Use the flag **`-f`** to specify which column(s) to return. Note that the column ID starts from 1 but not 0.

For example, returns the 1st and the 3rd columns from the file *example1.txt*.

```
$ cut -d ' ' -f 1,3 example1.txt
5 3.5
4 50
1 5.5
10 5
3 15
2 8
```

```
5 Apple 3.5
4 Chicken 50
1 Coke 5.5
10 Jelly 5
3 Chocolate 15
2 Milk 8
```

example1.txt

3.5 Remove adjacent duplicate lines

The **uniq** command removes the adjacent duplicate lines so that only one of the duplicated lines remains. Note that it only removes adjacent duplicates.

For example, remove adjacent duplicates in **example2.txt**

```
$ uniq example2.txt
Apple
Apple pie
Apple
Apple pie
```

```
Apple
Apple pie
Apple pie
Apple
Apple
Apple pie
```

example2.txt

3.6 Check Spelling

The **spell** command displays all **incorrect** words in a file.

```
$ spell example3.txt
beautiffful
happpy
todday
```

```
It's a beautiffful day!
I am so happpy todday.
```

example3.txt

Note: If your own Linux system does not have the command **spell**, please install it by:

1. Switch to root account.

   ```
   $ su
   ```

2. Install the **aspell** package

   Debian/Ubuntu:

   ```
   $ apt-get install aspell
   ```

   Red Hat/Fedora/CentOS:

   ```
   $ yum install aspell
   ```

3. Go back to your original user account

   ```
   $ exit
   ```

3.7 Difference between two line

The **`diff`** command outputs a description of how to transform the file in the 1st argument to the file in the 2nd argument.

Consider the following two files:

```
aaa
bbb
ccc
```

fileA.txt

```
eee
aaa
ddd
```

fileB.txt

The following command returns the process to transform *fileA* to *fileB*.

```
$ diff fileA.txt fileB.txt
0a1
> eee
2,3c3
< bbb
< ccc
---
> ddd
```

Explanations

`0a1` – To **a**dd a line after line **0** of *fileA*, the line to be added is line **1** of *fileB* (denoted by **`> eee`**). (Figure 5)

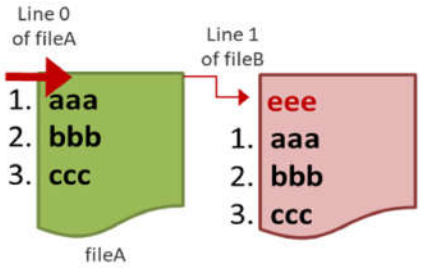


Figure 5

`2,3c3` – To **c**hange lines **2,3** of *fileA* to line **3** of *fileB*. The lines to be deleted are **bbb** and **ccc** (denoted by **`< bbb`** and **`< ccc`**); the line to be inserted is **ddd** (denoted by **`> ddd`**). (Figure 6)

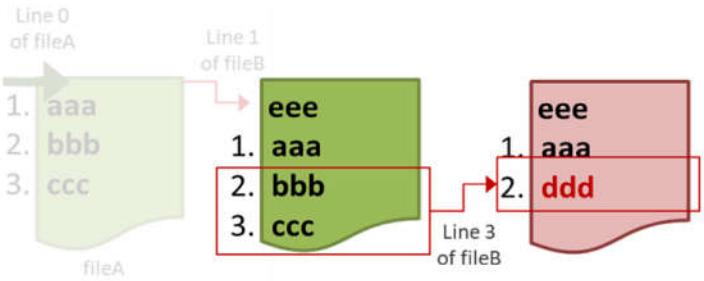


Figure 6

Another Example

The following command returns how to transform *fileB* to *fileA*.

```
$ diff fileB.txt fileA.txt
1d0
< eee
3c2,3
< ddd
---
> bbb
> ccc
```

Explanations

`1d0` – To **d**elete line **1** from *fileB*, and the files will then be in sync starting at line **0**. The line to be deleted is **eee** (denoted by **`< eee`**). (See Figure 7)

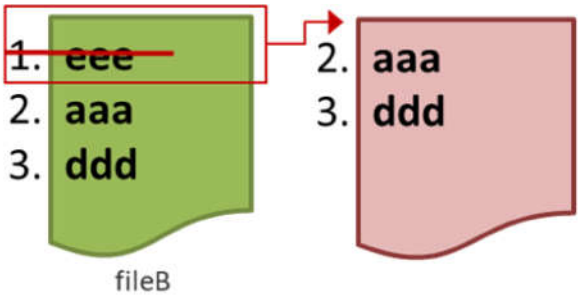


Figure 7

`3c2,3` –To **c**hange line **3** of *fileB* to line **2,3** of *fileA*. The line to be deleted is **ddd** (denoted by **`< ddd`**); the lines to be inserted are **bbb** and **ccc** (denoted by **`> bbb`** and **`> ccc`**). (See Figure 8)

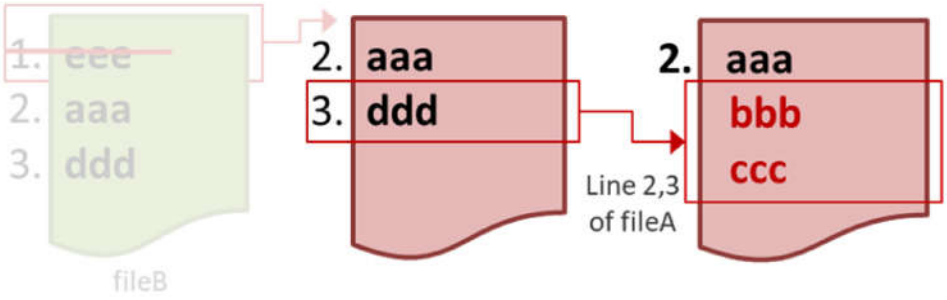


Figure 8

4. Standard I/O, File redirection, and Pipe

4.1 Standard I/O, File redirection

4.1.1 Standard input, Standard output, and Standard error

When using commands in the shell, the shell associates each process with some open files and references these open files by using numbers called file descriptors.

File descriptor	Files
0	Standard input (stdin) –The input file.
1	Standard output (stdout) – The output file.
2	Standard error (stderr) – The error output file.

Usually, when we execute a command, the output is printed on the screen. We can redirect the output to a file using the file **redirection operator >**.

For example, the following command stores the content of the directory into a file *files.txt*.

```
$ ls -l 1> files.txt
```

Or short form

```
$ ls -l > files.txt
```

The result of the above command is not displayed on the screen but redirected to the file *files.txt* instead. This is a very useful technique to create files storing the output of shell commands.

*If the above command is executed more than once, the system may say "**cannot overwrite existing file**". To solve this issue, you may manually remove the existing file first or use **>|** to force overwrite the file.

```
$ ls -l >| files.txt
```

Below shows the meaning of the redirection operators >, >>, < and <<.

Command	Explanations
`command 1> file` or `command > file`	Send standard output to *file*.
`command 1>> file` or `command >> file`	Append standard output to *file*.
`command 2> file`	Send standard error to *file*.
`command 2>> file`	Append standard error to *file*.
`command < fileA > fileB 2>fileC`	the command gets its input from *fileA* and sends output to *fileB*, error to *fileC*.

4.1.2 The > operator

Redirect the output of the directory listing into the files *result.txt*

```
$ ls -l 1> result.txt
```

Or short form

```
$ ls -l > result.txt
```

4.1.3 The >> operator

The operator **>>** is the same as **>** except it appends the content to a file instead of replacing the content of the.

```
$ wc data.txt >> result.txt
```

If you look at *result.txt*, the content of `wc data.txt` should be appended at the end of *result.txt*.

4.1.4 Redirect Standard Error

Sometimes a command may return an error message. For example, if you grep a file that does not exist, then the grep command returns an **error** message.

Note that the error messages are stored in the **standard error file** (the file with file descriptor 2)

Example 1

Suppose that the file *fileA* does not exist, if we just execute the command without file redirection, the standard error will be displayed on the screen.

```
$ grep 'abc' fileA
grep: fileA: No such file or directory
```

If we use file redirection for the standard error file (File descriptor: 2), the error will not be displayed on the screen. Here, the error will be redirected to a file named *error.txt* (Figure 9).

```
$ grep 'abc' fileA 2> error.txt
```

```
grep: fileA: No such file or directory
```

*Figure 9 Content of **error.txt** after execution*

Example 2

Redirect the standard error to the same location as where we redirect the standard output (`2>&1`).

```
$ cp fileA fileB 1> result.txt 2>&1
```

If fileA does not exist, a standard error is generated. Because of `2>&1`, the standard error is redirected to the same location as standard output *result.txt*.

4.1.5 The < operator

Similarly, if we have a program or command that accepts user inputs, we can redirect the file as input to the program or command by the **redirection operator <**.

Assume we have a C++ program named *add.cpp*. This program accepts two integer inputs and then output the sum.

```
//add.cpp
#include <iostream>
int main() {
   int a;
   int b;
   std::cin >> a;
   std::cin >> b;
   std::cout << a + b;
}
```

Don't panic! We will learn C++ in later modules.

To compile *add.cpp* into executable *add*.

```
$ g++ add.cpp -o add
```

Run *add*, then type in 2 integer values in the console. It will print the result (e.g. 3 + 4 = 7).

```
$ ./add
3 4
7
```

File redirection: Alternatively, we can run *add* with input from a file.

Use the **redirection operator <** to redirect the content of *input.txt (Figure 10)* into the *add* program

```
$ ./add < input.txt
7
```

```
3 4
```

Figure 10 input.txt

4.1.6 Combined Use

We can combine file input and output redirection.

Redirect the input input.txt (Figure 10) to the program add and redirect the program output to ouput.txt (Figure 11).

```
$ ./add < input.txt > output.txt
```

```
7
```

Figure 11 Content of ouput.txt after execution.

4.2 Pipe

Sometimes, we want to redirect the output of a program as the input of another program.

For example, to find all files created on Jan 26, we can use a temporary file to store the result of the **`ls`** command. Then, we can **`grep`** those lines containing the pattern "Jan 26", as follows.

```
$ ls -l > files.txt
$ grep "Jan 26" < files.txt
```

However, the method above is not good enough. It is because it creates an intermediate file *files.txt*. Note that *files.txt* could be very large if there are many directories and files under the current directory, but the result of **`grep`** could be just a few files or directories.

There is a more convenient way by using a pipe (i.e., the "|" symbol), which redirects the output of one program directly into the input of another program. (No intermediate files needed)

```
$ ls -l | grep "Jan 26"
```

In the above command, the "|" symbol means pipe, which redirects the output of the command **`ls -l`** directly into the input of the **`grep`** command.

Example 1

Sort the products in *data.txt* by their price, and store only the product name and product price in the file *result.txt* using one command (Figure 12).

```
$ sort -k3 -n data.txt | cut -d' ' -f2,3 > result.txt
```

The intermediate result is then piped into the input of the second command **`cut -d' ' -f2,3`**, which generates the final result. The final result is then redirected to a file named *result.txt*.

Illustration:

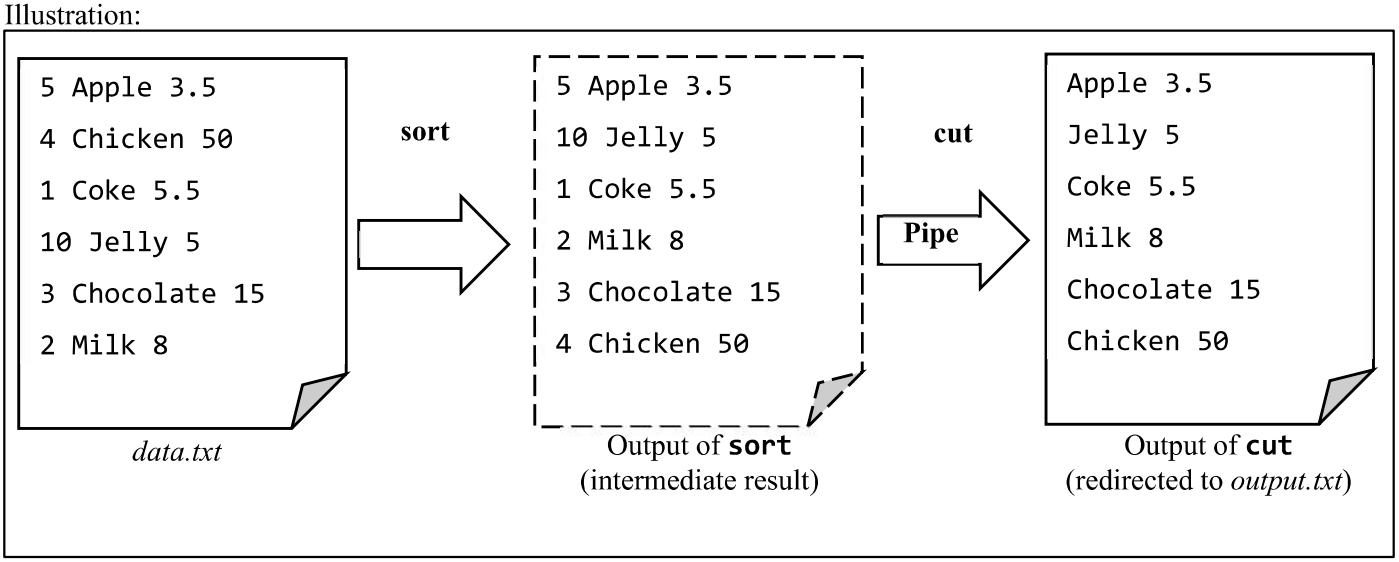


Figure 12 An example of using Pipe

Example 2

Find the file/directories in the current directories with execute (x) permission for user, group, and others.

The command to get the files and directories with the permission indicators

```
$ ls -l
total 190
-rwx--x--x.    1   kit  gopher   0   Sep  12 10:30  add.o
...
```

The files that all users, group, and others have execute permissions only if the 4th, 7th, and 10th position of indicators are "x". (i.e. "**---x--x--x**")

So, the command to return those files that are executable by users, group, and others using regular expression is:

```
$ grep -E '^...x..x..x' [the result of ls -l]
```

For your references:

- "^" indicates the matching start from the beginning of the line.
- "." indicates to match any single character.
- "^...x..x..x" indicates to match the lines from the beginning, return the line if the 4th, 7th, and 10th letters are "x".

Therefore, we can **pipe** the output of **ls -l** to the input of the **grep** command.

```
$ ls -l | grep -E '^...x..x..x'
```

Example 3

Suppose you want to remove some of the column(s) in a file then input it into a program.

Here, you have a file *mark.txt* that contain one-line data of student information and marks.

```
$ cat mark.txt
2011111111 John M 98.5 100 62.5 88 75.5
```

The format is [UID, name, gender, mark1, mark2, mark3, mark4, mark5]

And you have the following C++ file *mark.cpp*

```
//mark.cpp
#include <iostream>
int main() {
   double a1, a2, a3, a4, a5;
   std::cin >> a1 >> a2 >> a3 >> a4 >> a5;
   std::cout << "The overall mark the student get is: ";
   std::cout << (a1 + a2 + a3 + a4 + a5) / 5;
}
```

The problem is that *mark.cpp* only reads in assignment scores only, but *mark.txt* contains UID, name, and gender in the first few columns.

One possible way to pipe the data in *mark.txt* to the program is to cut the data in *mark.txt* so that it pipes only the assignment marks.

The result of cutting *mark.txt* is

```
$ cut -d' ' -f4-8 mark.txt
98.5 100 62.5 88 75.5
```

Compile the program and pipe the result to the program:

```
$ g++ mark.cpp -o mark
$ cut -d' ' -f4-8 mark.txt | ./mark > result.txt
```

The output of the program will be redirected to *result.txt*

```
$ cat result.txt
The overall marks the student get is: 84.9
```

5. Searching

5.1 Search for files / directories (find)

Sometimes you need to find files or directories with certain characteristics such as filename, size, permission, etc. The **find** command is a very powerful tool, which can drill right down through the system or just the directory you are looking for.

The format of **find** is

```
find [path] [-name] [-type]
```

- *path* is the actual path or directory where you want **find** to start drilling.
- *name* is the name of the file or directory to be searched.
- *type* (optional), -type f means to search for files only, and –type d means to search for directories only.

Example

Assume there are two files (*hello.txt*, *hello.cpp*) and one directory (*home*) in the current directory.

```
$ ls
hello/  hello.cpp  hello.txt
```

To search for a file *hello.txt* in the current directory and all its sub-directories.

```
$ find . -name "hello.txt" -type f
./hello.txt
```

Note: The dot (.) defines the **current directory**, as find will drill down starting from the current directory; the above command is to search for the file *hello.txt* in the current directory and all its subdirectories.

To search for files named begin with "*hello.*".

```
$ find . -name "hello.*" -type f
./hello.cpp
./hello.txt
```

To search for a directory in the current directory and all its any sub-directories.

```
$ find . -name "hello" -type d
./hello
```

find can do a lot more besides search by name. For example, **find** can search by file ownership, modification time, size, etc. Please refer to the references if you are interested in learning more about the command **find**.

5.2 Search inside a file (grep)

grep (Global Regular Expression Print) is used to search in a file and return the lines that match the pattern specified in a regular expression.

The format of **grep** is

grep -E '*regular expression*' *filename*

The most common way to use grep is to search for the lines consist of a given word, in that case, we do not need the flag -E.

E.g., search for lines containing '**hell**' in the file example1.txt (Figure 13).

```
$ grep 'hell' example1.txt
I am using the bash shell!
```

*grep is case sensitive! it only matches the "s**hell**" in the 2nd line but NOT the "**Hell**o" in the 1st line.

```
Hello how are you?
I am using the bash shell!
```

Figure 13 example1.txt

To specify more sophisticated matching patterns

1. We need to use a regular expression to match more complicated patterns.
2. Use the flag -E when using regular expression in grep.

Regular expression

Symbol	Meaning
`.`	Matching any **single character**.
`^`	Match the **beginning of the line** only.
`$`	Match the **end of the line** only.
`?`	A single character followed by an `?`, will match **zero or one occurrence**.
`+`	A single character followed by an `+`, will match **one or more occurrences**.
`*`	A single character followed by an `*`, will match **zero or more occurrences**.
`[]`	Character enclosed inside the `[ ]`will be matched. This can be a **single or range of characters**. You can use the "`-`" to include a range inclusively. E.g., instead of saying `[12345]`, use `[1-5]`.
`\`	Use this to **escape** the special meaning of a metacharacter. E.g., As "`.`" means matching any single character, we need to use "`\.`" to mean that we are matching a dot in a pattern.
`pattern {n}`	Match **`n` occurrence**s of the pattern.
`pattern {n,}`	Match **at least `n` occurrences** of the pattern.
`pattern {n,m}`	Match occurrences of the pattern **between `n` and `m`**.
`(ab){3}`	3 occurrences of the pattern '`ab`' . For example, `(ab){3}` will match "`ababab`", but not "`abbb`".

5.2.1 Match any single character

Use a dot (.) to match any character to search for patterns **any single character followed by "ell"** in *example1.txt (Figure 13)*.

```
$ grep -E '.ell' example1.txt
Hello how are you?
I am using the bash shell!
```

- We need to use the flag -E because '**`.ell`**' is a regular expression, which contains the regular expression symbol.
- With '**`.ell`**', even if the words "Cell", "cell", "bell", etc., will be matched. If you want to limit the single character to be only 'H' or 'h', then you need to use the square bracket '**`[Hh]ell`**'. We will have more examples of the use of squares in the following sections.

5.2.2 Match at the beginning and end of a line

Assume we have a file *example2.txt*

```
$ cat example2.txt
apple
pineapple
apple pie
```

Use '`^`' to match the pattern at the beginning of the line.

```
$ grep -E '^apple' example2.txt
apple
apple pie
```

```
$ grep 'apple' example2.txt
apple
pineapple
apple pie
```

Use '**`$`**' to match the pattern at the end of the line.

```
$ grep -E 'apple$' example2.txt
apple
pineapple
```

Use '`^`' and '**`$`**' together to match the exact content of a line.

```
$ grep -E '^apple$' example2.txt
apple
```

Use '`^`' and '**`$`**' to specify the exact content of a line and use 5 "**`.`**" to express "any 5 characters" to match a line that contains exactly 5 characters.

```
$ grep -E '^.....$' example2.txt
apple
```

5.2.3 The use of '?', '+' and '*'

The following 3 notations are used to specify the number of occurrences of the character immediately ahead of them:

?	A single character followed by ?, will match **zero or one occurrence** of the character.
+	A single character followed by +, will match **one or more occurrences** of the character.
*	A single character followed by *, will match **zero or more occurrences** of the character.

Example 1 (?)

Use '**?**' to match lines that are followed by **zero or one occurrence** of the character "p".

```
$ grep -E '^ap?' example3.txt
apple
ape
angel
```

```
apple
coco
cherries
orange
ape
angel
```

Figure 14 example3.txt

Explanations:

1. "**^**" force the matching start at the beginning of a line.
2. "**ap**ple" is returned as the **first two characters** "ap" matched the expression ("a" followed by one occurrence of "p").
3. "**ap**e" is returned as the **first two characters** "ap" matched the expression ("a" followed by one occurrence of "p").
4. "**a**ngel" is returned as the first character "a" matched the expression ("a" followed by zero occurrences of "p").

Example 2 (+)

Use '**+**' to match lines that are followed by **one or more occurrences** of the character "p".

```
$ grep -E '^ap+' example3.txt
apple
ape
```

Explanations:

1. "**app**le" is returned as the **first three characters** "app" matched the expression ("a" followed by two (satisfies one or more) occurrences of "p").
2. "**ap**e" is returned as the **first two characters** "ap" matched the expression ("a" followed by one (satisfies one or more) occurrence of "p").
3. "angel" is NOT returned because although it is "a" matched, it is NOT followed by any character "p". ("**+**" requires one or more occurrences)

Example 3 (*)

Use '*' to match lines that are followed by **zero or more occurrences** of the character "p".

```
$ grep -E '^ap*' example3.txt
apple
ape
angel
```

Explanations:

- "**app**le" is returned as the **first three characters** "app" matched the expression ("a" followed by two (satisfies zero or more) occurrences of "p").
- "**ap**e" is returned as the **first two characters** "ap" matched the expression ("a" followed by one (satisfies zero or more) occurrence of "p")
- "**a**ngel" is returned as the **first character** "a" matched the expression ("a" followed by zero (satisfies zero or more) occurrences of "p")

Example 4 (Combine '.' and '*')

Here, we want to match a character "a", followed by any number of characters, and then followed by "ge".

```
$ grep -E 'a.*ge' example3.txt
orange
angel
```

Explanations:

- "or**ange**" and "**ange**l" are returned as both lines have the substring "ange" , which is a character "a" followed by any number of characters (i.e., denoted by the expression ".*"), then followed by "ge".

Example 5 (use of "( )")

Use the parentheses "( )" to mark the substring.

Here, we want to match lines that the substring "co" occurs one or more times.

```
$ grep -E '(co)+' example3.txt
coco
```

Note: If we use the regular expression "**co***" then both "coco" and "cherries" will be returned because the "*****" applies to the character "o" only, which essentially means return lines with a character "c" followed by zero or more "o".

```
$ grep -E 'co*' example3.txt
coco
cherries
```

5.2.4 Match any value in a set

We can use the square bracket "**[ ]**" to specify matching any character in a set.

- **[0123456789]**, or **[0-9]** matches any single digit.
- **[A-Z]** matches any single capital letter.
- **[a-z]** matches any single small letter.
- **[A-Za-z]** matches any single letter (both capital letters and small letters).

Assume we have the file *example4.txt*.

```
Apple Juice HKD13
apple pie USD4
Banana phone HKD
```

To find lines containing "apple" or "Apple".

```
$ grep -E '[Aa]pple' example4.txt
Apple Juice HKD13
apple pie USD4
```

To find lines containing "HKD" and follow by zero or more numbers.

```
$ grep -E 'HKD[0-9]*' example4.txt
Apple Juice HKD13
Banana phone HKD
```

- **[0-9]** defines any digit from 0 to 9
- **[0-9]*** defines zero or more occurrences of any digit, which the 1st and 3rd rows in *example4.txt*.

5.2.5 Match a limited number of occurrences

We can use "**{ }**" to specify matching the number of occurrence that the pattern appears.

Assume we have the file *example5.txt*.

```
2April2013
30-1-2013
13December2013
```

To match the date in DayMonthYear format, where

- Day can be 1 or 2 digits
- Month is at least 3 letters
- Year is exactly 4 digits

Use the following regular expression:

```
$ grep -E '^[0-9]{1,2}[a-zA-Z]{3,}[0-9]{4}' example5.txt
2April2013
13December2013
```

- '**^**' force to match at the start of a line, and [0-9] specifies any digit.
- **[0-9]{1,2}** specifies at least 1 digit and at most 2 digits, which matches the **Day** part.
- **[a-zA-Z]{3,}** specifies at least 3 letters, which matches the **Month** part.
- **[0-9]{4}** specifies exactly 4 digits, which matches the **Year** part.

Note that **[a-zA-Z]** means any letters, you cannot write it in this way: **[a-z,A-Z]**. In that case, it indicates any letters plus a comma "**,**" are allowed.

5.2.6 Other useful Regular expression patterns

Regular expression	Meaning
`[a-z]*`	Any number of lower-case letters.
`^....$`	Lines containing 4 characters.
`abc.*abc`	Lines containing `abc`, followed by any number of characters, followed by `abc`.
`[0-9]{2}-[0-9]{2}-[0-9]{4}`	Date format `dd-mm-yyyy` (without validating the date)
`^.{n,m}$`	Lines with the length between n and m
`(bye)+`	Matches one or more occurrences of `bye`. E.g., `bye`, `byebye`, `byebyebye`, but not a null string.

6. Further Reading

We have introduced the Linux environment and the Bash shell. You will get familiar with them when you spend more time using them. The following webpages contain a very good introduction to working in Linux. You are highly recommended to read them once.

Linux tutorials:

3 Linux Tutorial: http://tldp.org/LDP/gs/node5.html
This website provides a comprehensive introduction of Linux commands

UNIX / Linux Tutorial: https://www.tutorialspoint.com/unix/index.htm
This website also provides comprehensive information of Linux commands, in a more organized way.

Command-line bootcamp: https://cli-boot.camp/
This website not only provides you with a brief introduction of Linux commands, but also includes an online Linux command line environment so you can try the commands within your browser.

Regular Expression:

Using Grep & Regular Expressions to Search for Text Patterns in Linux:
https://www.digitalocean.com/community/tutorials/using-grep-regular-expressions-to-search-for-text-patterns-in-linux
This website gives you more examples of using Regular Expressions in the "grep" command.

7. References

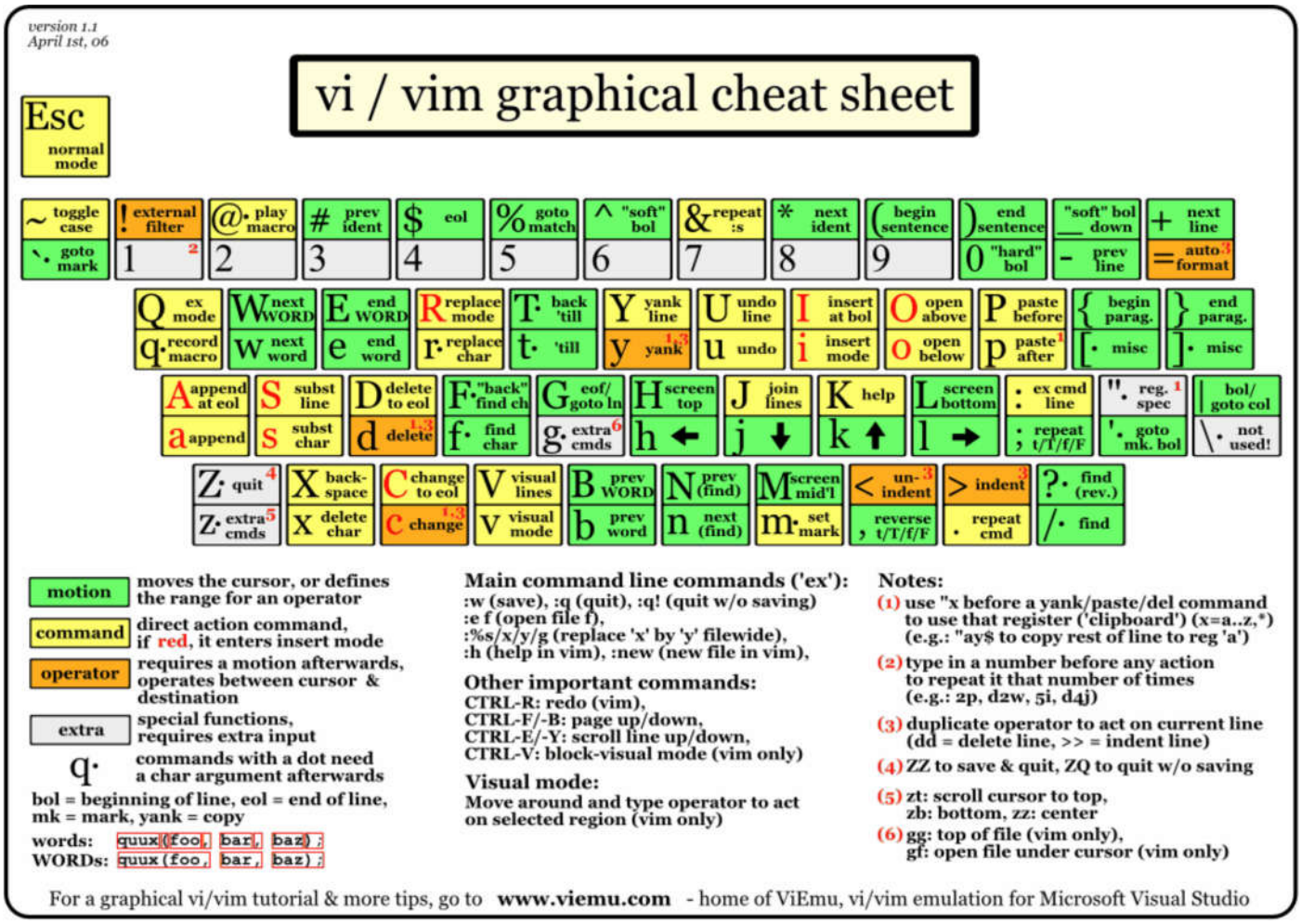
- Part 1. A Practical Guide to Linux(R) Commands, Editors, and Shell Programming, Mark Sobell. Prentice Hall PTR
- Chapter 1. File security and permission. *LINUX &UNIX Shell Programming.* Mr. David Tansley, Addison Wesley.
- An A-Z Index of the Bash command line for Linux. http://ss64.com/bash/
- UNIX – Lesson 017 – `chmod` command in symbolic-mode and in absolute-mode. http://tipsandtricks4it.wordpress.com/2010/01/25/lesson-017/
- Chmod Online tutorial: http://ss64.com/bash/chmod.html
- Linux.org tutorial on file permission: http://www.linux.org/article/view/file-permissions-chmod
- Understanding 'diff' command. http://tarique21.wordpress.com/2008/11/06/understanding-diff-command/
- Chapter 2. Using `find` and `xargs`. *LINUX & UNIX Shell Programming.* Mr. David Tansley, Addison Wesley.
- Online tutorial for `find`: http://ss64.com/bash/find.html
- A Beginner's Guide to Grep: Basics and Regular Expressions: http://www.linuxforu.com/2012/06/beginners-guide-gnu-grep-basics-regular-expressions/
- Online tutorial for regular expression: http://www.regular-expressions.info/
- Chapter 7. Introducing regular expressions, Chapter 8. The `grep` family. *LINUX &UNIX Shell Programming.* Mr. David Tansley, Addison Wesley.

8. What can I do if I cannot follow the notes?

- Ask the student TAs during the online sessions.
- Read the reference materials for more detailed explanations.
- Post the questions on Moodle – We have set up a forum for each self-learning module. Please post your questions there, and we will answer your question shortly.
- Join our online consultation hours. Please send us an email in advance so that we can better schedule the consultation time for you.

9. Appendices

9.1 vi cheat sheet



Graphical vi-vim Cheat Sheet and Tutorial. (n.d.). Retrieved from http://www.viemu.com/a_vi_vim_graphical_cheat_sheet_tutorial.html