Binaries, refers to files that can be executed, similar to executables in Windows.

Directory, this is the same thing as a folder in Windows. A directory provides a way of organizing files, usually in a hierarchical manner.

Home, each user has their own /home directory, and this is generally where files you create will be saved by default.

Short cut for opening terminal is CTRL + ALT + T.

To get help on something use:

$ aircrack-ng --help

Here –-help can be -h or -? also but not sure if it will work or not for every command.

To go into sudo mode: $ sudo su

to see the cool logo of kali or any Linux distribution use

$ sudo apt-get install neofetch

$ neofetch

$ printenv SHELL

Use this to get the current shell the terminal is using; like bash or zsh. To switch to bash use $ sudo bash.

$ passwd

To change your password, use $ passwd, it will first ask for old password then new password and will ask you to re-type your new password.

$ id & uname

id is used to get the id name. and uname is used to get the OS name. $ id -u -n, to get the operating system name. $ uname -s -r, to get info in verbose. $ uname -v, to get running processes

$ whoami

If you've forgotten whether you're logged in as root or another user, you can use the whoami command to see which user you're logged in as.

$ ps

The ps command is used to display information about process running on the machine.

$ top

top stands for "table of process", it is like task manager. To get resource usage; top is used to get the top n running process. Like $ top -n 3 will give top 3 running process.

$ df

df is used to get info on mounted file system, like which drives are connected and all. $ df -h is used to get the info in human readable format.

$ date

Will return current date and time as a string.

$ expr

Is like a calculator, if you run $ expr 4 + 5 it will return 9, but if you run,

$ expr 4+5 it will return 4+5 not it sums. Also, if you run $ expr 4 +5 or

$ expr 4+ 5 will return error. So take care of spaces using expr. However, expr cannot natively handle decimal places:

$ expr 1 + 2.5 will return error. To overcome that issue, we use bc.

$ bc

Stands for 'basic calculator'.

Call without any argument and shell will provide you space below to write your calculation. When you enter 5 + 7 there and hit enter it will return 12 but not exit interactive mode it will again wait for your input you can again write you calculation and hit enter as many times you like. To exit write quit, the shell will exit interactive mode and will be ready to take new commands.

It will also work with decimal values.

If you don't want to open the calculator right there you could use piping. Like so:

$ echo "5 + 7.5" | bc

Will return 12.5. bc also has a scale argument for how many decimal places.

$ echo "10 / 3" | bc

By default, it will return 3. But you want control over how many decimals to be used use scale like so:

$ echo "scale=3; 10 / 3" | bc

Will return 3.333 (up to three decimal places). Note the use of ; is to separate lines in terminal.

Sometimes bc with –l option is also use for complex calculation, like so bc –l where l means load math library.

$ clear

Clear is a helper command. It is used to clear the terminal.

An *absolute path* is like a latitude and longitude: it has the same value no matter where you are.
A *relative path*, on the other hand, specifies a location starting from where you are: it's like saying "20 kilometres north".

The shell decides if a path is absolute or relative by looking at its first character: If it begins with /, it is absolute. If it *does not* begin with /, it is relative.

csv is a file type and it stands for “comma separated values”.

One of the shell's power tools is tab completion. If you start typing the name of a file and then press the tab key, the shell will do its best to auto-complete the path. For example, if you type sea and press tab, it will fill in the directory name seasonal/ (with a trailing slash). If you then type a and tab, it will complete the path as seasonal/autumn.csv.

If the path is ambiguous, such as seasonal/s, pressing tab a second time will display a list of possibilities. Typing another character or two to make your path more specific and then pressing tab will fill in the rest of the name.

$ pwd

Stands for “present working directory” (where you are).

$ ls

Stands for “listing” (what’s there).

If you add the names of directories, it will list their contents. For example, $ ls /home/repl shows you what's in your starting directory (usually called your home directory). Some useful flags of ls are:

- –a: will list all things, including filenames starting with ‘.’.
- -l: long list formatting, it will give name, size, date created, permissions, etc.
- -t: will sort by time, newest first.

  ls flags can also be used together like:
  $ ls –l –a and $ ls –la will do the same thing.

$ dir

It will list in files separated with '\' instead of space.

$ cd

Stands for "change directory" (where to go). If you want to go to folder with a name with space in it use 'folder_name' with cd.

If you want to get back to your home directory /home/repl, you can use the command cd /home/repl.

More often, though, you will take advantage of the fact that the special path .. (two dots with no spaces) means "the directory above the one I'm currently in". If you are in /home/repl/seasonal, then cd .. moves you up to /home/repl. If you use cd .. once again, it puts you in /home. One more cd .. puts you in the *root directory* /, which is the very top of the filesystem. (Remember to put a space between cd and .. - it is a command and a path, not a single four-letter command.)

A single dot on its own, ., always means "the current directory", so ls on its own and ls . do the same thing, while cd . has no effect (because it moves you into the directory you're currently in).

One final special path is ~ (the tilde character), which means "your home directory", such as /home/repl. No matter where you are, ls ~ will always list the contents of your home directory, and cd ~ will always take you home.

It can be used in combination like cd ~/../.

cd with no arguments will take you to home directory.

cd – will take you to just previous directory which you were in.

$ locate

locate, followed by a keyword denoting what it is you want to find, this command will go through your entire file system and locate every occurrence of that word.

Some more advanced commands for searching a file are: whereis, which and find.

$ find . -name "file to <find.extention>", where . means start searching from where you are for case insensitive use -iname flag.

you can search with specific directory, filename, etc

$ find /home -type f -name *.c

here find is searching in /home, search type is file, name is like anything.c.

$ cp

Stands for “copy”.

```
cp original.txt duplicate.txt
```

you can also include path name here.

creates a copy of original.txt called duplicate.txt. If there already was a file called duplicate.txt, it is overwritten.

If the last parameter to cp is an existing directory, then a command like:

```
cp seasonal/autumn.csv seasonal/winter.csv backup
```

copies *all* of the files into that directory. Here seasonal/autumn.csv and seasonal/winter.csv are files with path and backup is not a command it is the folder where you want to copy these files provided all these files and folder are in same directory as you are in. cp –v flag will provide verbose (what is it doing) during copying. This –v flag also works with rm, mv etc.

$ touch

It is used to create empty files. $ touch file1.txt file2.py [file] will create file1.txt, file2.py and other names you give. It can also create files without any extension.

$ mv

Stands for “move” (move file)

It handles its parameters the same way as cp

```
mv autumn.csv winter.csv ..
```

moves the files autumn.csv and winter.csv from the current working directory up one level to its parent directory (because .. always refers to the directory above your current location).

mv can also be used to rename files. If you run:

```
mv course.txt old-course.txt
```

then the file course.txt in the current working directory is "moved" to the file old-course.txt. notice here because no folder name is given at last so it renames the file.

mv treats directories the same way it treats files: if you provide directory name instead of file name it renames directory name.

One warning: just like cp, mv will overwrite existing files. If, for example, you already have a file called old-course.txt, then the command shown above will replace it with whatever is in course.txt.

$ rm

Stands for “remove” (to delete).

For example:

```
rm thesis.txt backup/thesis-2017-08.txt
```

removes both thesis.txt and backup/thesis-2017-08.txt.

The shell doesn’t have a trash can. So, when rm is used it is permanently deleted.

Notice here with rm only file names are required of those you want to remove nothing else.

If you try to rm a directory, the shell prints an error message telling you it can't do that, primarily to stop you from accidentally deleting an entire directory full of work. Instead, you can use a separate command called rmdir. For added safety, it only works when the directory is empty, so you must delete the files in a directory *before* you delete the directory. But if you want to delete directory and its contents simultaneously you can use –r flag like $ rm –r sampledirectory will remove the sample directory even it has files in it.

$ mkdir

maybe Stands for “make directory”.

Syntax $ mkdir <directory_name>.

to create a directory into another directory, use relative path. For example if you are in a directory where yearly directory is already present then $ mkdir yearly/2017 will create directory 2017 in yearly without moving to yearly directory.

You will often create intermediate files when analyzing data. Rather than storing them in your home directory, you can put them in /tmp, which is where people and programs often keep files they only need briefly. (Note that /tmp is immediately below the root directory /, *not* below your home directory.)

```
mv ~/people/agarwal.txt scratch
```

see here how ~ is used, it always means home directory.

$ cat

Stands for “concatenate” (prints the contents of the file).

Type only cat <file_name_with_extension>. However, cat can also be used to create small files, like so: $ cat > filename will open interactive mode and wait for you to start entering content for the file. When you are done press ^d to exit interactive mode and a new file will be created. You can also create new file by using >> in place of > but there is a twist. If you will again use $ cat > the samefile the previous content of the file will be deleted and new will be added. To overcome this use >> instead of > to append content.

$ less

Type only less <file_name_with_extension>.

When you less a file, one page is displayed at a time; you can press spacebar to page down or type q to quit.

If you give less the names of several files, you can type :n (colon and a lower-case 'n') to move to the next file, :p to go back to the previous one, or :q to quit.

Example:

```
less seasonal/spring.csv seasonal/summer.csv
```

will print both the files, if you press space bar it will scroll down a bit. Then if you type :n it will move to next file, and if you type :q it will quit the operation and will be ready to take commands.

$ more

It is similar to less, but in addition it shows how much percent of the total file is shown.

$ head

Type only head <file_name_with_extension>.

As its name suggests, it prints the first few lines of a file (where "a few" means 10).

You won't always want to look at the first 10 lines of a file, so the shell lets you change head's behaviour by giving it a command-line flag (or just "flag" for short). If you run the command:

```
head -n 3 seasonal/summer.csv
```

head will only display the first three lines of the file. If you run head -n 100, it will display the first 100 (assuming there are that many), and so on. Here also you can put +[NUM] after –n for specifying from which line number you want to print. Similarly, for byte count use –. Actually you can directly put number line from which you want output, like so: head -3 samplefile.txt.

If you want to display specified characters of a file, use –c flag.

A flag's name usually indicates its purpose (for example, -n is meant to signal "number of lines"). Command flags don't have to be a - followed by a single letter, but it's a widely-used convention.

$ ls -R

In order to see everything underneath a directory, no matter how deeply nested it is, you can give ls the flag -R (which means "recursive"). If you use ls -R in your home directory, you will see something like this:

```
backup          course.txt      people          seasonal

./backup:

./people:
agarwal.txt

./seasonal:
autumn.csv      spring.csv      summer.csv      winter.csv
```

This shows every file and directory in the current level, then everything in each sub-directory, and so on.

$ ls –F

ls has another flag -F that prints a / after the name of every directory and a * after the name of every runnable program.

-R and –F flags can be used simultaneously and the order doesn't matter.

$ man

Stands for manual (To find out what commands do).

For example, the command man head brings up this information:

```
HEAD(1)                 BSD General Commands Manual                HEAD(1)

NAME
     head -- display first lines of a file

SYNOPSIS
     head [-n count | -c bytes] [file ...]

DESCRIPTION
     This filter displays the first count lines or bytes of each of
     the specified files, or of the standard input if no files are
     specified.  If count is omitted it defaults to 10.

     If more than a single file is specified, each file is preceded by
     a header consisting of the string ``==> XXX <==" where ``XXX"
     is the name of the file.

SEE ALSO
     tail(1)
```

man automatically invokes less, so you may need to press spacebar to page through the information and :q to quit.

$ tail

Gives output for the last part of the file. Here also you can use –n and +[NUM] like in head. Like tail –n +2 will start from second line. tail –n N will without – sign will grab last N lines. Similarly, we can do for characters like tail –c N will grab last N characters.

$ nl

Stands for “number line”.

To display a file with line numbers. Syntax: $ nl sample.txt

The same thing which nl does can be done by cat –b sample.txt. remember the above commands do not count spaces as lines. To count spaces as well use cat –n sample.txt.

$ cut

head and tail let you select rows from a text file. If you want to select columns, you can use the command cut.

Most common example is:

```
cut -f 2-5,8 -d , values.csv
```

which means "select columns 2 through 5 and columns 8, using comma as the separator". cut uses -f (meaning "fields") to specify columns and -d (meaning "delimiter") to specify the separator. You need to specify the latter because some files may use spaces, tabs, or colons to separate columns. in the above example –d is specifying , (comma) so it prints columns which are separated by , (comma) if : was specified with –d it uses : to separate values. If delimiter is tab use –d $'\t'. if delimiter is space use –d " ". '' can also be used instead of "".

Another useful flag is –c which means characters. cut –c N means only output Nth character from each line. If you run cut –c N,M then it will give output of Nth and Mth character from each line. If I run cut –c N-M it will give output of Nth character through Mth character both included. To print the characters from Nth position to the end of the line use: cut –c N-. Notice N- is used only.

$ history

history will print a list of commands you have run recently. Each one is preceded by a serial number to make it easy to re-run particular commands: just type !55 to re-run the 55th command in your history (if you have that many). You can also re-run a command by typing an exclamation mark followed by the command's name, such as !head or !cut, which will re-run the most recent use of that command.

$ grep

grep selects lines according to what they contain. In its simplest form, grep takes a piece of text followed by one or more filenames and prints all of the lines in those files that contain that text. For example, grep bicuspid seasonal/winter.csv prints lines from winter.csv that contain "bicuspid". Other examples of using grep are:

```
$ grep ‘a’ fruits.txt
```

Will match everything which has ‘a’ in it.

```
$ grep ‘[abcd]’ fruits.txt
```

Will match everything which has either ‘a’, ‘b’, ‘c’ or ‘d’ in fruits.txt. It’s something like RegEx.

grep's more common flags:

- -c: print a count of matching lines rather than the lines themselves
- -h: do not print the names of files when searching multiple files
- -i: ignore case (e.g., treat "Regression" and "regression" as matches)
- -l: print the names of files that contain matches, not the matches
- -n: print line numbers for matching lines
- -v: invert the match, i.e., only show lines that don't match
- -q: supress all normal output. Doesn’t return the line if text is present in that file.

Another useful flag for grep is –e it is used to pass multiple search patterns in a single line. Like you want to search for lines which either contain ‘hello’ or ‘bye’. The command will be like:

```
$ grep –e ‘hello|bye’
```

The same can be achieved by egrep like so:

```
$ egrep ‘hello|bye’
```

$ sed

Stands for ‘stream editor’. It is basically used for find and replace actions. The syntax for sed command is:

```
$ sed ‘s/pattern/change/’ file.txt
```

Where the pattern is replaced with change. If you want to replace the nth occurrence, then use:

$ sed ‘s/pattern/change/n’ file.txt

$ sed ‘s/pattern/change/g’ file.txt

Here g means global, it will replace every occurrence.

$ sed ‘s/pattern/change/gI’ file.txt

This is for global with case insensitive.

$ sed ‘s/pattern/{&}/gI’ file.txt

This if for covering all patterns in {}.

sed -r 's/[0-9]{4}[ ]/**** /g'

this is to mask first 4 digits of a number with *.

$ awk

Syntax:

```
$ awk options 'selection _criteria {action }' input-file > output-file
```

By default:

$ awk ‘{print}’ file.txt

Will work same like $ cat file.txt. If you want to print only those lines which match a particular patter use:

$ awk ‘/pattern/ {print}’ file.txt

Will only print those lines which has pattern in it.

For each record i.e. line, the awk command splits the record delimited by whitespace character by default and stores it in the $n variables. If the line has 4 words, it will be stored in $1, $2, $3 and $4 respectively. Also, $0 represents the whole line. This is like similar to RegEx.

Suppose if a line is Jett has 30 coins to print only first part and last part use:

$ awk '{print $1,$4}' file.txt, means to print the first and forth part of the line.

$ paste

paste that can be used to combine data files instead of cutting them up.

It has some useful flags:

1. –d:

How to store a commands output in a file?

For example, if you run:

```
head -n 5 seasonal/summer.csv > top.csv
```

nothing appears on the screen. Instead, head's output is put in a new file called top.csv. This > can also be used with pipe commands. Like:

```
cut -d , -f 2 seasonal/*.csv | grep -v Tooth > teeth-only.txt
```

but if you try to use it in middle of pipe command like this:

```
cut -d , -f 2 seasonal/*.csv > teeth-only.txt | grep -v Tooth
```

then all of the output from cut is written to teeth-only.txt, so there is nothing left for grep and it waits forever for some input. If command like this is given:

```
> result.txt head -n 3 seasonal/winter.csv
```

It will work as usual, result of head will be stored in result.txt

You can take a look at that file's contents using cat:

```
cat top.csv
```

The greater-than sign > tells the shell to redirect head's output to a file. It isn't part of the head command; instead, it works with every shell command that produces output.

This technique can be used if you want lines 3-5 from a file. So first you would use head to select first 5 lines from that file and redirect it to a new file using >, then tail to get the last 3 lines from that new file.

But,

Using redirection to combine commands has two drawbacks:

1. It leaves a lot of intermediate files lying around (like top.csv).
2. The commands to produce your final result are scattered across several lines of history.

The shell provides another tool that solves both of these problems at once called a *pipe*. Once again, start by running head:

```
head -n 5 seasonal/summer.csv
```

Instead of sending head's output to a file, add a vertical bar and the tail command without a filename:

```
head -n 5 seasonal/summer.csv | tail -n 3
```

The pipe symbol tells the shell to use the output of the command on the left as the input to the command on the right.

You can chain any number of commands together with this pipe (|).

$ tr

It stands for “translate”. It does not translate languages, it translates characters. It is helpful to change one character to another and it is case sensitive. Notice it does not translate words it is only of characters. Its syntax is:

```
$ tr [OPTION] SET1 [SET2]
```

It has the following options to play with:

1. -c: complements the set of characters in string. i.e., operations apply to characters not in the given set
2. -d: delete characters in the first set from the output.
3. -s: replaces repeated characters listed in the set1 with single occurrence
4. -t: truncates set1

Say if you want to convert all lower cases to upper cases the command will be: $ echo “hello world” | tr “[a-z]” “[A-Z]” another more logical command will be: $ echo “hello world” | tr “[:lower:]” “[:upper:]”.

To translate white-spaces to tabs: $ echo "Welcome To The World" | tr [:space:] '\t'.

To translate braces to parenthesis: $ tr '{}' '()' newfile.txt

To squeeze repetition of character (make only one occurrence of that character at once): $ echo "Welcome To Home" | tr -s [:space:] will give output: Welcome To Home.

To delete specified characters: $ echo "Welcome To world" | tr -d 'w' will remove w not W.

To remove all the digits from the string: $ echo "my ID is 73535" | tr -d [:digit:] or tr –d [0-9].

$ wc

The command wc (short for "word count") prints the number of characters, words, and lines in a file. You can make it print only one of these using -c, -w, or -l respectively.

Most shell commands will work on multiple files if you give them multiple filenames. For example, you can get the first column from all of the seasonal data files at once like this:

```
cut -d , -f 1 seasonal/winter.csv seasonal/spring.csv seasonal/summer.csv seasonal/autumn.csv
```

But typing the names of many files over and over is a bad idea: it wastes time, and sooner or later you will either leave a file out or repeat a file's name. To make your life better, the shell allows you to use wildcards to specify a list of files with a single expression. The most common wildcard is *, which means "match zero or more characters", like Regex. Using it, we can shorten the cut command above to this:

```
cut -d , -f 1 seasonal/*
```

or:

```
cut -d , -f 1 seasonal/*.csv
```

like in Excel.

The shell has other wildcards as well, though they are less commonly used:

- ? matches a single character, so 201?.txt will match 2017.txt or 2018.txt, but not 2017-01.txt.
- [...] matches any one of the characters inside the square brackets, so 201[78].txt matches 2017.txt or 2018.txt, but not 2016.txt.
- {...} matches any of the comma-separated patterns inside the curly brackets, so {*.txt, *.csv} matches any file whose name ends with .txt or .csv, but not files whose names end with .pdf. * can also match zero characters.

$ sort

sort puts data in order. By default, it does this in ascending alphabetical order, but the flags -n and -r can be used to sort numerically and reverse the order of its output, while -b tells it to ignore leading blanks and -f tells it to fold case (i.e., be case-insensitive). Pipelines often use grep to get rid of unwanted records and then sort to put the remaining records in order. It also has an option to redirect the output to another file. You could do $ sort inputfile.txt > newfile.txt but with –o option you could

do $ sort –o newfile.txt inputfile.txt. It is also able to sort a table on the basis of any column number using –k option. For example, using –k 2 will sort on the second column and –k 2n for second column numerically. To check if file is already sorted or not use –c flag. To remove duplicates while sorting use –u option. One very important flag is –t, it is used to identify the fields to separate the values. Like while sorting to specify how to choose value see this example: $ sort –k 2 –n –t$'\t'. Here this example is used to separate fields as tab. You can also use any other delimiter in –t tab.

$ uniq

It is used to remove duplicate lines. But only *adjacent* duplicate lines.

For example:

If a file contains:

```
2017-07-03
2017-07-03
2017-08-03
2017-08-03
```

then uniq will produce:

```
2017-07-03
2017-08-03
```

but if it contains:

```
2017-07-03
2017-08-03
2017-07-03
2017-08-03
```

then uniq will print all four lines.

It has a useful flag –c. It tells how many times a line was repeated by displaying a number as a prefix with the line. Another flag is –d which only prints the repeated lines not the lines which aren't repeated. By default, the comparisons done are case sensitive but with –i option case will be ignored. –u will only print only unique lines.

$ nano <newfilenamewithextension>

Will open Linux text editor

$ echo

$ echo dfgsfdh or $ echo 'dffgsgs' both will work. It's like Linux version of print like in Python.

But if you use read with just echo, the read input will be taken from next like. If you want to take input from the same line use echo –n.

Like other programs, the shell stores information in variables. Some of these, called **environment variables**, are available all the time. Environment variables' names are conventionally written in upper case, and a few of the more commonly-used ones are shown below.

Variable	Purpose	Value
HOME	User's home directory	/home/repl
PWD	Present working directory	Same as pwd command
SHELL	Which shell program is being used	/bin/bash
USER	User's ID	repl

To get a complete list (which is quite long), you can type set in the shell.

If you try to use it to print a variable's value like this:

```
echo USER
```

it will print the variable's name, USER.

To get the variable's value, you must put a dollar sign $ in front of it. Typing

```
echo $USER
```

The other kind of variable is called a **shell variable**, which is like a local variable in a programming language.

To create a shell variable, you simply assign a value to a name:

```
training=seasonal/summer.csv
```

without any spaces before or after the = sign. Once you have done this, you can check the variable's value with:

```
echo $training

seasonal/summer.csv
```

Shell variables are also used in **loops**, which repeat commands many times. If we run this command:

```
for filetype in gif jpg png; do echo $filetype; done
```

it produces:

```
gif
jpg
png
```

Notice these things about the loop:

1. The structure is for ...variable... in ...list... ; do ...body... ; done
2. The list of things the loop is to process (in our case, the words gif, jpg, and png).
3. The variable that keeps track of which thing the loop is currently processing (in our case, filetype).
4. The body of the loop that does the processing (in our case, echo $filetype).

Notice that the body uses $filetype to get the variable's value instead of just filetype, just like it does with any other shell variable. Also notice where the semi-colons go: the first one comes between the list and the keyword do, and the second comes between the body and the keyword done.

This bellow example will clarify

```
for filename in seasonal/*.csv; do echo $filename; done
```

It prints:

```
seasonal/autumn.csv
seasonal/spring.csv
seasonal/summer.csv
seasonal/winter.csv
```

People often set a variable using a wildcard expression to record a list of filenames. For example, if you define datasets like this:

```
datasets=seasonal/*.csv
```

you can display the files' names later using:

```
for filename in $datasets; do echo $filename; done
```

This loop prints the second line of each data file:

```
for file in seasonal/*.csv; do head -n 2 $file | tail -n 1; done
```

a loop can contain any number of commands. To tell the shell where one ends and the next begins, you must separate them with semi-colons:

```
for f in seasonal/*.csv; do echo $f; head -n 2 $f | tail -n 1; done

seasonal/autumn.csv
2017-01-05,canine
seasonal/spring.csv
2017-01-25,wisdom
seasonal/summer.csv
2017-01-11,canine
seasonal/winter.csv
2017-01-03,bicuspid
```

Controls for nano editor:

- Ctrl + K: delete a line.
- Ctrl + U: un-delete a line.
- Ctrl + O: save the file ('O' stands for 'output'). You will also need to press Enter to confirm the filename!
- Ctrl + X: exit the editor.

How can I record what I just did?

1. Run history.
2. Pipe its output to tail -n 10 (or however many recent steps you want to save).
3. Redirect that to a file called something like figure-5.history.

put the following command in a file called headers.sh:

```
head -n 1 seasonal/*.csv
```

This command selects the first row from each of the CSV files in the seasonal directory. Once you have created this file, you can run it by typing:

```
bash headers.sh
```

This tells the shell (which is just a program called bash) to run the commands contained in the file headers.sh, which produces the same output as running the commands directly.

Notice bash is needed to run the script file.

A file full of shell commands is called a ***shell script**, or sometimes just a "script" for short. Scripts don't have to have names ending in .sh, but this lesson will use that convention to help you keep track of which files are scripts.

You can also use pipe commands in scripts.

You can also redirect the output of bash script to another file like this:

```
bash all-dates.sh > dates.out
```

will extract the dates from the seasonal data files and save them in dates.out.

A script that processes specific files is useful as a record of what you did, but one that allows you to process any files you want is more useful. To support this, you can use the special expression $@ (dollar sign immediately followed by at-sign) to mean "all of the command-line parameters given to the script".

For example, if unique-lines.sh contains sort $@ | uniq, when you run:

```
bash unique-lines.sh seasonal/summer.csv
```

the shell replaces $@ with seasonal/summer.csv and processes one file. If you run this:

```
bash unique-lines.sh seasonal/summer.csv seasonal/autumn.csv
```

it processes two data files, and so on.

As a reminder, to save what you have written in Nano, type Ctrl + O to write the file out, then Enter to confirm the filename, then Ctrl + X to exit the editor.

Note that in Nano, "copy and paste" is achieved by navigating to the line you want to copy, pressing CTRL + K to cut the line, then CTRL + U twice to paste two copies of it.

Loops in shell scripts

Shell scripts can also contain loops. You can write them using semi-colons, or split them across lines without semi-colons to make them more readable:

```
# Print the first and last data records of each file.

for filename in $@

do

    head -n 2 $filename | tail -n 1

    tail -n 1 $filename

done
```

(You don't have to indent the commands inside the loop, but doing so makes things clearer.)

The first line of this script is a comment to tell readers what the script does. Comments start with the # character and run to the end of the line. Your future self will thank you for adding brief explanations like the one shown here to every script you write.

***you use sudo to give poweruser to yourself to undo this use sudo –k.

$ sudo su

For going in root mode

After this you don't need to type sudo again and again

$ crunch 8 8 0123456789

Will create a list of random character look up

In Debian-based Linux distributions, which include Kali and Ubuntu,
the default software manager is the Advanced Packaging Tool, or apt,
whose primary command is apt-get. You can use apt-get to download and install new software packages, but you can also update and upgrade software with it.

Before downloading a software package, you can check whether the
package you need is available from your repository, which is a place
where your operating system stores information.
The apt tool has a search function that can check whether the package is available. The syntax is straightforward:

$ apt-cache search *keyword*

This will give numerous files having the keyword. When you find your package in the repository, you can use apt-get to download the software, followed by the keyword install. Like so:

$ apt-get install *packagename*

To remove a software, use apt-get with remove option, followed by the name of the software to remove.

$ apt-get remove *package*

Remember, **updating** is not the same as **upgrading**: updating simply updates the list of packages available for download from the repository, whereas upgrading will upgrade the package to the latest version in the repository.

To update your individual system use: $ apt-get update this will search through all the packages on your system and check whether updates are available. If so, the updates will be downloaded.

To upgrade the existing packages on your system, use $ apt-get upgrade. Because upgrading your packages may make changes to your software, you must be logged in as root or use the sudo command before entering the command.

If a particular software is only available on GitHub, you can download the whole package by $ git clone URL where URL is the URL where the package is located.

To view running process on your system use $ ps, it will not show much information. Running the ps command without any options lists the process started by the currently logged-in user and what process are running on the terminal. This doesn't show

PID: process ID, the Linux kernel assigns a PID to each process sequentially, as the processes are created.