

Bash Shell Basics

This is a concise refresher on the basic commands available with the Bash shell. It is designed as a prerequisite module to the Supercomputing User Training offered by [Pawsey Supercomputing Research Centre](#), Perth (Australia).

You can access this reference document both as a [web page](#) and as a [PDF document](#). For a more detailed tutorial on the Bash shell, see [The Unix Shell](#) by the [Software Carpentries](#).

Outline

1. [Definitions](#)
 2. [Hands-on](#)
 - 2.1. [Handling directories](#)
 - 2.2. [Handling files](#)
 - 2.3. [Using environment variables](#)
 - 2.4. [Editing and visualising text files](#)
 - 2.5. [Manipulating outputs and text files](#)
 - 2.6. [More commands](#)
 - 2.7. [Keyboard tricks](#)
-

Definitions

- **Command Line Interface (CLI)**: a type of interface to use a computer, that requires the user to type textual **commands** and read their textual outputs by means of a prompt. Some commands may require optional or mandatory **arguments**.
- **Shell**: a program with a prompt providing a command line interface, and an interpreter to execute commands. There are different implementations of a shell, using different **shell languages** for the commands.
- **Bash**: one of the most popular shell languages in the Linux world.
- **Shell script**: a text file containing a sequence of shell commands. It can be executed via the shell, and is useful to increase reproducibility of a workflow.
- Main advantages of using a shell:
 - Productivity (lots of actions with few keystrokes)
 - Task automation
 - Reproducibility (using shell scripts)
- **Text editor**: a program used to open and edit textual files; some of them are designed for use within a command line interface, such as **nano**, **vi** and **emacs**.

NOTE: in all the examples below, by convention commands are identified by lines starting with \$, which represents the prompt (not to be typed). Other lines represent command

outputs. Also note how # denotes a comment line in Bash.

Hands-on

Handling directories

First, navigate to the home directory using `cd`; here, `~` refers to the home, and can be omitted if used alone:

```
$ cd ~  
$ # or also  
$ cd
```

Check the path of the **current working directory** with `pwd`:

```
$ pwd  
/home/username
```

Create a new directory with `mkdir`:

```
$ mkdir training
```

Enter it, and check where you are:

```
$ cd training  
$ pwd  
/home/username/training
```

The current directory can be referred to using a single dot, `.`:

```
$ cd .  
$ pwd  
/home/username/training
```

The **parent directory** can be referred to using a double dot, `..`:

```
$ cd ..  
$ pwd  
/home/username  
$ cd training
```

Create two more directories inside `training`, create one more into the first one of these, and then access the first one:

```
$ mkdir dir1 dir2
```

```
$ mkdir dir1/one-more-dir
$ cd dir1
```

A **full path** does not depend on the current working directory, and starts with a slash, /, e.g.: /home/username/training.

A **relative path** depends on the current working directory, and starts with a directory name, a . or a .., e.g.: one-more-dir, ./one-more-dir, ../dir2.

Go back to the training parent directory:

```
$ cd ..
$ pwd
/home/username/training
```

Handling files

Now let us play with files. Make new empty files using touch:

```
$ touch file1
$ touch file2 file3
$ touch dir1/file4
```

The command is actually designed for altering timestamps of existing files; however it will create an empty file, if the provided filename does not match an existing one.

Check the content of a directory with ls:

```
$ ls
dir1 dir2 file1 file2 file3
$ ls dir1
file4 one-more-dir
```

Get more details about files and directories with ls -l:

```
$ ls -l
total 8
drwxr-x--- 3 username username 4096 Aug 26 14:10 dir1
drwxr-x--- 2 username username 4096 Aug 26 14:10 dir2
-rw-r----- 1 username username    0 Aug 26 14:10 file1
-rw-r----- 1 username username    0 Aug 26 14:10 file2
-rw-r----- 1 username username    0 Aug 26 14:10 file3
```

To make a copy of a file with another name use cp:

```
$ cp file1 copy1
```

To copy a directory use `cp -r`:

```
$ cp -r dir1 copydir1
```

To rename a file use `mv`:

```
$ mv copy1 another-file
```

To move a file to another directory use `mv` again, but now with the second argument as an existing directory:

```
$ mv another-file dir2/
```

Use `rm` to remove a file:

```
$ rm file3
```

Use `rm -r` to remove a directory:

```
$ rm -r copydir1
```

Multiple files and/or directories can be selected using the **wildcard** characters `?` and `*`. The question mark, `?`, means any value of a single character:

```
$ ls file?  
file1 file2
```

The asterisk, `*`, means any value of any number of characters:

```
$ ls f*  
file1 file2
```

When the `*` also includes directories, `ls` will output their content, too:

```
$ ls *  
file1 file2  
  
dir1:  
file4 one-more-dir  
  
dir2:  
another-file
```

Using environment variables

You can assign a **value**, *i.e.* a string of characters, to a named environment **variable**. We call **shell environment** the whole set of variables defined in the current shell session.

Variables are typically lost when the current shell session is closed.

Assign a variable using the equal operator, =:

```
$ HELLO="world"
```

The value can be reused later in the same shell session by referring to the corresponding variable. For instance, use `echo` to just display the variable value:

```
$ echo $HELLO  
world
```

To make the variable accessible inside sub-processes, you need to **export** it:

```
$ export BYE="moon"
```

A process could be a program, a shell script, or just another Bash shell. Let us verify the latter case. Start a new Bash shell with `bash`, and then `exit` it when done:

```
$ bash  
$ echo $HELLO  
  
$ echo $BYE  
moon  
$ exit  
exit
```

You can use a variable to store a directory path:

```
$ pwd  
/home/username/training  
$ MYDIR="/home/username/training"
```

And then use it:

```
$ echo $MYDIR  
/home/username/training  
$ ls $MYDIR  
dir1 dir2 file1 file2
```

There is a method to store the output of a command inside the variable: encapsulating the command within the syntax `$()`. For instance, use such syntax to capture and store the

output of pwd:

```
$ MYDIR_AGAIN="$(pwd)"  
$ echo $MYDIR_AGAIN  
/home/username/training
```

Every shell session automatically defines a number of environment variables, some of which can be quite useful. Among others, \$HOME contains the path to your home directory, \$USER contains your username, and \$PATH has the list of paths where the shell looks for executable programs and scripts:

```
$ echo $HOME  
/home/username  
$ echo $USER  
username  
$ echo $PATH  
/usr/local/bin:/usr/bin:/bin
```

Editing and visualising text files

Let us now use the text editor nano to create, edit and save a text file. First, start the editor with a new empty text file called text:

```
$ nano text
```

Now, type the following in the editor:

```
Line 1  
Fancy line 2  
Super line 3  
Extra line 4  
Last line 5
```

To save this text, press Ctrl-O (Control + O), then confirm with Enter. To exit the editor, press Ctrl-X.

Visualise the full contents of the file from the shell using cat:

```
$ cat text  
Line 1  
Fancy line 2  
Super line 3  
Extra line 4  
Last line 5
```

For long files, the command less allows showing the contents one screen at a time. The

next screen can be accessed using Space or Ctrl-F (forward), the previous screen using Ctrl-B (backward). Quit the less screen with q. You can trial this with the file you just created, although the output will most likely fit in a single screen:

```
$ less text
Line 1
Fancy line 2
Super line 3
Extra line 4
Last line 5
text lines 1-5/5 (END)
$ # quit by typing `q`
```

Use head/tail to visualise the first/last lines of a file. By default 10 lines are displayed, but a custom number can be specified using a specific command option:

```
$ head -1 text
Line 1
$ tail -1 text
Last line 5
```

Manipulating outputs and text files

Use wc to get the line, word and character count of a text:

```
$ wc text
5 14 58 text
```

You can see that the text file has 5 lines, 14 words and 58 characters (including newlines).

Use grep to filter text lines matching a specific string pattern. Search for the string Super in the text file created above:

```
$ grep Super text
Super line 3
```

Commands can be connected together to feed the output of a command as the input for the following command. To connect commands use the **pipe operator**, |:

```
$ echo "Hello world from Pawsey." | wc
      1      4     25
```

Redirection operators can be used to connect a file with the output or input of a command.

Use > to redirect a command output into a file (file will be overwritten):

```
$ echo "Hello world from Pawsey." >hello
$ cat hello
Hello world from Pawsey.
$ echo "Hello again." >hello
$ cat hello
Hello again.
```

Use >> to append a command output into a file (file will not be overwritten):

```
$ echo "Bye bye from Pawsey." >>bye
$ echo "Bye bye again." >>bye
$ cat bye
Bye bye from Pawsey.
Bye bye again.
```

Use < to feed the contents of a file as command input:

```
$ wc < bye
2  7 36
```

Other useful commands for file and output manipulations include cut, tr, sort, sed and awk.

More commands

Use hostname to output the name of the computer/server where the shell session is open:

```
$ hostname
setonix-01
```

Use which to get the full path of a program:

```
$ which grep
/usr/bin/grep
```

Some commands are not programs (see next command to know more):

```
$ which cd
which: no cd in (/usr/local/bin:/usr/bin:/bin)
```

Use type to get information about the typology of command:


```
$ type grep
grep is /usr/bin/grep
$ type cd
cd is a shell builtin
```

Sometimes it is useful to group together a set of files and/or directories within a single archive file, which is typically called tarball and assigned the extension `.tar`. The `tar` program is used to this end.

Create a tarball containing all files and/or directories (with their contents) whose name starts with `file` or `dir`:

```
$ tar cf archive1.tar file* dir*
```

Visualise its contents:

```
$ tar tf archive.tar
file1
file2
dir1/
dir1/one-more-dir/
dir1/file4
dir2/
dir2/another-file
```

Now create a directory `expand`, enter it, unpack the tarball there, and check the directory contents:

```
$ mkdir expand
$ cd expand
$ tar xf ../archive1.tar
$ ls
dir1  dir2  file1  file2
cd ..
```

If you also want to compress the tarball to reduce its size, add the option `z` to the commands above, and use the conventional extension `.tar.gz`:

```
$ tar czf archive2.tar.gz file* dir*
$ ls -l archive*
-rw-r----- 1 username username 20480 Aug 26 15:59 archive1.tar
-rw-r----- 1 username username   581 Aug 26 16:03 archive2.tar.gz
```

The latter tarball takes 581 bytes instead of 20480. Note that compression rates may vary significantly depending on the file contents.

Other useful Bash syntax constructs include for/while loops and if/elif/else conditionals.

Keyboard tricks

Use Up-arrow and Down-arrow to browse the shell history of commands:

```
$ # hit Up-arrow once
$ ls -l archive*
$ # hit Up-arrow again
$ tar czf archive2.tar.gz file* dir*
$ # hit Down-arrow once
$ ls -l archive*
```

Note you will need to hit Enter to actually execute the selected command, as usual.

Tab completion is a handy feature that can reduce typing and make shell usage more convenient. Start typing a command, then hit Tab to see available commands whose name start with the typed characters:

```
$ # type without hitting Enter
$ host
$ # hit Tab twice
host      hostid      hostname  hostnamectl
$ hostn
$ # hit Tab again
$ hostname
```

Again, to execute the selected command hit Enter, after eventually having edited it or added more arguments.

If you need to stop what you are typing, or a running program, and get back to an empty shell within the same session, use Ctrl-C:

```
$ # type without hitting Enter
$ hostname
$ # hit Ctrl-C
$ hostname^C
$
```

This is all for this Bash shell refresher. You can close the shell session with exit!
