Text in **`<angled brackets>`** are templates and not literal.
Starred commands are specific to Bash or Debian/Ubuntu, or require installing a package to run.

Basic Bash commands

Working with directories

- **`pwd`** prints the working directory
- **`ls <directory>`** lists files in **`<directory>`**, or the current directory if no directory is provided
  - Add the **`-a`** flag to include hidden files (i.e. files whose names start with .)
  - Add the **`-l`** flag to list files in long listing format (which prints some metadata like which users and groups own them and who can read, write, and execute them)
- ★ **`tree <directory>`** lists files in **`<directory>`** in a hierarchical, tree-like format
- **`cd <directory>`** changes the current directory to **`<directory>`**
  - Use **`.`** for the current directory, **`..`** for the parent directory and **`-`** for the previous directory
- **`mkdir <directory>`** creates a directory called **`<directory>`**
- **`rmdir <directory>`** deletes **`<directory>`** if it is empty

Working with files

- **`file <file>`** prints what kind of file **`<file>`** is
- **`stat <file>`** prints **`<file>`**'s metadata, including when it was last modified and accessed as well as its size
- **`touch <file>`** creates a file called **`<file>`**, or updates the last accessed date of **`<file>`** if it already exists
- ★ **`vim <file>`** opens **`<file>`** in the Vim text editor (usage not discussed in detail here)
- **`cat <...files>`** prints the contents of **`<files>`** to the terminal
  - Add the **`-n`** flag to number the lines
- **`head -n <number> <file>`** prints the first **`<number>`** lines of **`<file>`**, or 10 lines if no number is provided
- **`tail -n <number> <file>`** prints the last **`<number>`** lines of **`<file>`**, or 10 lines if no number is provided
  - Add the **`-f`** flag to display new additions to the end of **`<file>`** in real time
- **`sort <file>`** prints every line of **`<file>`** to the terminal like **`cat`**, but in lexicographic sorted order
  - Add the **`-n`** flag to sort by numerical order instead
  - Add the **`-M`** flag to sort by month order instead
- **`grep -E <regex> <file>`** searches for **`<regex>`** in **`<file>`**
  - The **`grep`** command accepts many arguments; look up more if interested
- **`mv <file> <directory>`** moves **`<file>`** to **`<directory>`**, or renames **`<file>`** to **`<directory>`** if **`<directory>`** doesn't exist, e.g. **`mv <current_file_name> <new_file_name>`**
- **`cp <source> <destination>`** copies **`<source>`**'s data to **`<destination>`**

    - `<source>` and `<destination>` can be files or directories
        - If `<source>` is a directory, add the `-r` flag to recursively copy its contents
    - If `<destination>` doesn't exist, it gets created
- `rm <file>` deletes `<file>`
    - Add the `-rf` flags to recursively force delete all files in a directory, though be VERY careful running this command- it is undoable
- `tar -czvf <tarball_name>.tar.gz <...files, ...directories>` creates a gzip-compressed archive file named `<tarball_name>.tar.gz` containing the provided `<files>` and/or `<directories>`
- `tar -xzvf <tarball_name>.tar.gz` extracts the gzip-compressed archive file `<tarball_name>.tar.gz`

Executing commands and files

- `history` lists recently-run commands
    - **Up arrow key** brings up the previous command in the command history
    - `!!` runs the most recently-run command
- `type -a <command>` prints the path where `<command>`'s executable is located if it is an external command, or else prints that `<command>` is a built-in command
    - Running an external command creates a new process; running a built-in command does not
- `man <command>` opens the manual page for `<command>`
- `sudo <command>` runs `<command>` as the root user (sudo stands for "superuser do")
    - The root user has elevated privileges, and some commands require these privileges to run
    - You will be prompted to enter the current user's password
- `<command> &` runs `<command>` in the background
- `<command1>; <command2>` runs `<command1>` and then `<command2>`
- `crontab -e` opens the current user's crontab in an editor
    - Add entries (called cronjobs) formatted as `<minute> <hour> <day> <month> <week> <command>` to automatically run commands at scheduled times
        - For example, `30 2 * * * /home/user/script.sh` executes `/home/user/script.sh` every day at 2:30 AM
- `chmod <ugo><+-=><rwx> <file>` changes which users can read, write, and execute `<file>`
    - `sudo` is necessary if you are not an owner of `<file>`
    - `u` updates the user owner's permissions, `g` updates the group owner's permissions, `o` updates others' permissions, and omitting updates all users' permissions
    - `+` adds bits, `-` removes bits, and = sets bits
    - `r` is the read bit, `w` is the write bit, and `x` is the execute bit
- `./<file>` executes `<file>` from the current directory if it is executable
    - Do not include `./` if `<file>` is an absolute path
- `source <file>` executes `<file>` in the current shell session (rather than a subshell)

Working with processes

- **`jobs`** lists all processes in the current shell session. Note that each has a job ID (which is different from a process ID!) in brackets
    - Add the **`-l`** flag to print process IDs as well
- **`ps -ef`** lists all processes in the system and status information about each of them
    - The **`ps`** command accepts many arguments; look up more if interested
- ★ **`htop`** displays information about processes like **`ps`**, but in real time
- **ctrl+z** sends a SIGSTP signal, suspending the currently running process
- **ctrl+c** sends a SIGINT signal, forcefully terminating the currently running process
- **`fg %<job_ID>`** foregrounds/continues the execution of the process with job ID **`%<job_ID>`** in the foreground, or the most recently suspended process if no job ID is provided
- **`bg %<job_ID>`** backgrounds/continues the execution of the process with job ID **`%<job_ID>`** in the background, or the most recently suspended process if no job ID is provided
- **`kill <process_ID>`** sends a SIGTERM signal, terminating the process with the provided process or job ID if possible
- **`killall <process_name>`** sends a TERM signal, terminating the process(es) with the provided name if possible
    - If you run this as the root user, be VERY careful to not accidentally stop an important process
- **`exit`** terminates the process that executes it; in an interactive shell, that process is the shell itself

Managing users and groups

- ★ **`sudo adduser <user>`** creates a new user whose username is **`<user>`**
- **`cat /etc/passwd`** lists all users in the system and some information about each of them
- **`id <user>`** prints **`<user>`**'s ID, username, primary group ID, and groups they belong to, or the current user's if no user is provided
    - When a user is created, a private group with the same name is also created which that user is added to. This group becomes the user's primary group, and it helps prevent other users from modifying the user's files without permission
- **`passwd <user>`** changes **`<user>`**'s password
- **`su - <user>`** switches the current user to **`<user>`** after entering **`<user>`**'s password
- **`ssh <user>@<hostname>`** securely connects to the server at **`<hostname>`** as user **`<user>`**
    - **`<hostname>`** can be an IP address or URL
    - Add the **`-p <port>`** flag to specify which port to connect to (default is 22)
    - If no SSH key (usage not discussed in detail here) is found for **`<user>`** at **`<hostname>`**, you will be prompted to enter the **`<user>`**'s password
- **`sudo usermod -L <user>`** locks **`<user>`**'s account
- **`sudo usermod -U <user>`** unlocks **`<user>`**'s account

- ★ **`sudo deluser <user>`** deletes **`<user>`**'s account
    - ○ Add the **`--remove-home`** flag to delete **`<user>`**'s home directory as well
- ★ **`sudo addgroup <group>`** creates a new group whose name is **`<group>`**
- ● **`cat /etc/group`** lists all groups in the system and some information about each of them
- ★ **`sudo adduser <user> <group>`** adds **`<user>`** to **`<group>`**
- ★ **`sudo deluser <user> <group>`** removes **`<user>`** from **`<group>`**
- ★ **`sudo delgroup <group>`** deletes **`<group>`**
- ● **`sudo chown <user>:<group> <file>`** changes the owners of **`<file>`** to be user **`<user>`** and group **`<group>`**

Installing packages

- ★ **`apt search <package>`** searches for packages whose name or description contains **`<package>`**
- ★ **`sudo apt install <package>`** installs **`<package>`** and its dependencies
- ★ - **`apt list --installed`** lists installed packages
- ★ **`apt show <package>`** prints information about **`<package>`**
- ★ **`dpkg -L <package>`** lists all files installed by **`<package>`**
- ★ **`sudo apt update`** fetches the latest package lists from repositories
- ★ **`sudo apt upgrade`** installs the latest versions of packages already installed
- ★ **`sudo apt remove <package>`** uninstalls **`<package>`** but keeps its configuration files
- ★ **`sudo apt purge <package>`** uninstalls **`<package>`** and deletes its configuration files

Other

- ● **`echo <string>`** prints **`<string>`**
- ● **`date`** prints the current date
- ● **`clear`** clears the terminal
- ★ **`bc`** enters the bash calculator which can perform floating-point arithmetic
    - ○ Run **`scale=<number>`** inside the calculator to set the number of digits after the decimal point
    - ○ Run **`quit`** inside the calculator to exit
- ● **`sleep <number>`** waits **`<number>`** seconds
- ● **`df -h`** lists how much storage is free on each mounted disk
- ● **`du -h`** lists how much storage is used by each directory
- ★ **`wget <URL>`** sends a request for the resource at **`<URL>`** and writes the response to a file
    - ○ Add the **`-r`** flag to recursively download the resources at **`<URL>`**
- ★ **`curl <URL>`** sends a request for the resource at **`<URL>`** and prints the response

Navigating the terminal

- ● You can use wildcards such as **`*`** and **`\w`** and it will expand to all matches. ex: **`rm *.txt`**
- ● **tab key** autocompletes a command
- ● **ctrl+a** moves the cursor to the beginning of the command
- ● **ctrl+e** moves the cursor to the end of the command
- ● **ctrl+arrow keys** move the cursor backward or forward one word in the terminal

- **ctrl+l** clears the terminal
- **ctrl++** increases the terminal's font size
- **ctrl+-** decreases the terminal's font size
- ★ You can run multiple terminal sessions in one terminal window
    a. **`tmux`** creates another terminal session
    b. **ctrl+b** and then **%** splits the terminal window vertically
    c. **ctrl+b** and then **ctrl+arrow key** moves the cursor between terminal sessions
    d. **`tmux ls`** lists active terminal sessions
    e. **`tmux kill-server`** kills all terminal sessions

Shell operators

Unary operators

- **`true`** always evaluates to true (exit status **`0`**)
- **`false`** always evaluates to false (exit status **`1`**)
- **`-e <file>`** checks whether **`<file>`** exists
- **`-d <directory>`** checks whether **`<directory>`** exists and is a directory
- **`-f <file>`** checks whether **`<file>`** exists and is a regular file (i.e. not a directory, socket, or device)
- **`-s <file>`** checks whether **`<file>`** exists and is not empty
- **`-r <file>`** checks whether **`<file>`** exists and is readable
- **`-w <file>`** checks whether **`<file>`** exists and is writable
- **`-x <file>`** checks whether **`<file>`** exists and is executable

Binary operators

- **`<expression1> && <expression2>`** is a conditional and
- **`<expression1> || <expression2>`** is a conditional or
- **`<number1> -eq <number2>`** is a numeric equal to
- **`<number1> -ne <number2>`** is a numeric not equal to
- **`<number1> -gt <number2>`** is a numeric greater than
- **`<number1> -ge <number2>`** is a numeric greater than or equal to
- **`<number1> -lt <number2>`** is a numeric less than
- **`<number1> -le <number2>`** is a numeric less than or equal to

Input/output redirection

- **`<command> > <file>`** writes the output (i.e. stdout) of **`<command>`** into **`<file>`**
- **`<command> 2> <file>`** writes the error output (i.e. stderr) of **`<command>`** into **`<file>`**
- ★ **`<command> &> <file>`** writes both normal and error output of **`<command>`** into **`<file>`**
    - **`<command> &> /dev/null`** redirects both normal and error output of **`<command>`** to basically a void, discarding it
- **`<command> >> <file>`** appends the output of **`<command>`** to **`<file>`**
- **`<command> < <file>`** uses the contents of **`<file>`** as input (i.e. stdin) for **`<command>`**
- **`<command> < <file1> > <file2>`** uses the contents of **`<file1>`** as input for **`<command>`** and writes the output into **`<file2>`**

- **`<command1> | <command2>`** pipes (i.e. passes) the output of **`<command1>`** (run in a subshell) to **`<command2>`** (run in a different subshell)

Variables

Setting, viewing, and using variables

- **`<var>=<value>`** initializes a local environment variable named **`<var>`** with value **`<value>`**
    - Local environment variables are variables that are only accessible in the process that created them. Their names are typically lowercase
    - Note that you cannot put any spaces around **`=`**
    - If **`<value>`** is a string, it only needs to be enclosed with quotation marks if the string contains a space
    - **`<value>`** can also be an array, e.g. **`(1 2 3 4)`**, though arrays aren't commonly used
- **`export <var>`** makes variable **`<var>`** a global environment variable, making it accessible in child processes
    - Global environment variables are variables that are accessible in the current shell session and child processes (such as subshells). Their names are typically UPPERCASE
    - Changing the value of **`<var>`** in a child process does not change its value in the parent process
- **`env <var>`** prints the value of global environment variable **`<var>`**, or all global environment variables if no global variable is provided
- **`set`** prints the values of all environment variables, both global and local, that are defined in the current process
- **`$<var>`** expands variable **`<var>`** to the value that is bound to it
    - If **`<var>`** is an array, by default it only expands to the first element of the array
- **`unset <var>`** deletes variable **`<var>`**
    - **`unset`**ting **`<var>`** in a child process does not **`unset <var>`** in the parent process

Built-in variables

- **`$PATH`** is a colon-separated list of directories where the shell looks for commands
    - If you want to run a command whose executable is not one of the directories in **`$PATH`**, you can modify **`PATH`** to include the directory containing the command's executable so that the command can be run
- **`$HOME`** is the current user's home directory
- **`$PS1`** is the primary shell command line interface prompt string
- **`$IFS`** is the internal field separator characters
- **`$?`** is the exit status of the last command
    - By convention, **`0`** is success and anything else (**`1-255`**) is failure
- **`$#`** is the number of arguments
- **`$@`** is all arguments, where each argument is separated by one space
- **`$<number>`** is the **`<number>`**'th argument

        - Argument `0` is the name of the command/shell script/function itself, so the first argument provided to it would be `1`

More on variable expansions

- ``<var>=`<expression>` `` sets `<var>` to the output of `<expression>` (run in a subshell), which could be a command or another expression that evaluates to a value
    - For example, `<expression>` could be `echo hello`, which evaluates to `hello`
- `<var>=$(<expression>)` is equivalent to ``<var>=`<expression>` ``
- `<var>=$((<math_expression>))` sets `<var>` to the result of `<math_expression>`
    - This only supports integer arithmetic; run `bc` for floating-point arithmetic
- `${<var>}` expands `<var>` to the value that is bound to it
    - Preferred over omitting the curly braces since this removes ambiguity in some situations
    - Can use this to expand an array to a particular element in the array (i.e. `${<array>[<index>]}`)
    - Can use this to expand an array to all of the elements in the array (i.e. `${<array>[*]}`)
- ★ `${<var>:<lower_index>:<upper_index>}` gets a substring of `<var>` starting from `<lower_index>` until, but not including, `<upper_index>`
    - `:<upper_index>` can be omitted, in which case the substring will include up to the end of `<var>`

Quoting

- Single quotes (i.e. `''`) keeps every character between the single quotes literally as is
- Double quotes (i.e. `""`) keeps every character between the double quotes literally as is except for variable expansions (i.e. `$<some_variable>`), which it expands
- Commands can be enclosed in back ticks (i.e. ``` `` ```), and will expand to the result of the commands
    - For example, ``for i in `ls`; do...``

Control flow

Evaluating conditional expressions

- `test <expression>` returns `true` (exit status `0`) if `<expression>` is true, or `false` (non-zero exit status) otherwise
    - For example, `<expression>` could be `5 -lt 3`, which evaluates to `false`
- `[ <expression> ]` is equivalent to `test <expression>`
    - Note that you need a space between `<expression>` and the square brackets
    - You can chain `[ <expression> ]`'s with `&&` and `||` operators, e.g. `[ <expression1> ] && [ <expression2> ]`
    - `! [ <expression> ]` negates `[ <expression> ]`
        - Note that you need a space between `!` and `[`
- ★ `[[ <expression> ]]` is like `[ <expression> ]` but it allows for more string operators, such as `>` and `<` for string comparison, and pattern matching

★ **`(( <expression> ))`** is like **`[ <expression> ]`** but it allows for more number operators, such as **`>`** and **`<`** for number comparison

If statements

```
if <test_expression1>; then
      <commands1>
elif <test_expression2>; then
      <commands2>
else
      <default_commands>
fi
```

For loops

```
for <var> in <list>; do
      <commands>
done
```

- **`<list>`** can be:
  - A wildcard pattern that expands to matching pathnames, e.g. **`*.txt`**
  - A space-delimited sequence of tokens, e.g. **`Michigan Ohio Indiana`**
    - Can also obtain using command substitution, e.g. **`$(seq 1 10)`**
- **`break`** exits the loop early
- **`continue`** ends the current iteration and jumps to the next iteration
- Bash also introduces an alternative C-style for-loop syntax, but that isn't shown here

While loops

```
while <test_expression>; do
      <commands>
done
```

Until loops

```
until <test_expression>; do
      <commands>
done
```

Functions (Bashism)

```
<function_name> () {
      <commands>
}
```

- A function can be called with two arguments using **`<function_name> <arg1> <arg2>`**
  - You can extract the values of **`<arg1>`** and **`<arg2>`** in the function body from **`$1`** and **`$2`** respectively
- **`return`** exits the function and specifies its exit status

Case statements

```
case <expression> in
      <value1>)
            <commands1>
            ;;
      <value2>)
            <commands2>
            ;;
      *)
            <default_commands>
            ;;
esac
```

Shell scripts

Shell scripts contain commands that are run when the script is executed.
Before the commands, the first two lines of shell scripts should include:

- **`#!/bin/bash`**
  - This tells your shell that when executing the script, it should use Bash to execute it. Specifically, **`#!`** is called a shebang, and the text that follows it **`/bin/bash`** is the path where the bash shell's executable is located. Replace **`bash`** with a different shell such as **`zsh`** if you're using a different shell
- **`set -Eeuo pipefail`**
  - This is Bash-specific. It makes the script's exit status the exit status of the first failing command if it encounters an error while executing

Executing shell scripts

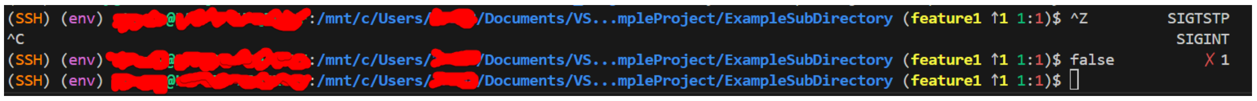
- **`ls -l <file>`** prints the read-write-executable bits of **`<file>`**
- **`chmod +x <file>`** makes **`<file>`** executable
- **`./<file>`** executes **`<file>`**

Example

```
      #!/bin/bash
      set -Eeuo pipefail
      if [ "$#" -ne 2 ]; then
          echo "Usage: $0 <num1> <num2>"
          exit 1
      fi
      num1=$1
      num2=$2
      res=$(bc << EOF
      scale=4
      $num1 / $num2
      EOF
      )
      echo "$res"
```

Shell configuration

I configured my shell to a certain extent by modifying `~/.bashrc`. There's a lot you can do with it, but I customized mine so that if my current directory has a git repository, the command prompt shows what git branch I'm in, how many commits the branch has, how many files are staged, and how many files are untracked.



```
(SSH) (env) [redacted]@[redacted]:/mnt/c/Users/[redacted]/Documents/VS...mpleProject/ExampleSubDirectory (feature1 ↑1 1:1)$ ^Z          SIGTSTP
^C                                                                                                                                        SIGINT
(SSH) (env) [redacted]@[redacted]:/mnt/c/Users/[redacted]/Documents/VS...mpleProject/ExampleSubDirectory (feature1 ↑1 1:1)$ false          ✗ 1
(SSH) (env) [redacted]@[redacted]:/mnt/c/Users/[redacted]/Documents/VS...mpleProject/ExampleSubDirectory (feature1 ↑1 1:1)$ ▯
```

GitHub repository at: https://github.com/Racekid16/bashrc

Extra (for fun)

Here is a bit of Bash code (called a fork-bomb) that you shouldn't run:

```
:(){ :|:& };:
```

If we format this code, it looks like this:

```
:() {
      : | :
      &
};
:
```

Explanation:

- **:** is actually the name of a function here. This may seem strange because it's likely that in most programming languages that you know, **:** can't be used in variable names. But in Bash, it's perfectly legal syntax
- **()** is part of what denotes that **:** is a function
- **{** and **}** enclose the function body
- **: | :** calls **:** and pipes the outputs of that function call to another function call to **:**. Note that simply writing **:** suffices to call the function since it doesn't take in any arguments. Since we're calling **:** inside the function definition for **:**, it's recursive
- **&** makes the code run in the background
- **;** is necessary to distinguish commands because all of the code is on one same line; in the formatted version it could be excluded
- The final **:** calls **:** outside of its function body, starting the fork bomb