Text in **<angled brackets>** are templates and not literal.
A “Bashism” is a feature specific to the Bash shell and not Unix-like shells in general.

Basic unix commands

Working with directories

- **pwd** prints the working directory
- **ls** lists files in the current directory. Add the **-a** flag to include hidden files (such as ones whose names start with .)
- **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>**

Printing to the terminal

- **echo <string>** prints **<string>** to the terminal
- **date** prints the current date to the terminal

Working with files

- **cat <file>** prints the contents of **<file>** to the terminal (provide more file arguments to concatenate all the provided files)
- **touch <file>** creates a file called **<file>** or updates the last accessed date of **<file>** if it already exists
- **stat <file>** shows when **<file>** was last modified and accessed as well as **<file>**’s size.
- **rm <file>** deletes **<file>** (you can also delete all files in a directory with **rm -rf <directory>**)
- **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>**)

Working with processes

- **jobs** lists all current and suspended processes. Note that each job has an id.
- **Ctrl+z** suspends the currently running process
- **Ctrl+c** forcefully terminates the currently running process
- **fg** foregrounds/continues the execution of the most recently suspended processes in the foreground
- **bg** backgrounds/continues the execution of the most recently suspended process in the background
- **kill** kills a process.
- Note that **fg**/**bg**/**kill** can take **%<job_id>** as an argument to specify that you want to foreground/background/kill that particular job. Example: **kill %1**

Other

- **man <command>** brings up the manual for **<command>**
- **sleep <number>** waits **<number>** seconds
- **source <file>** runs **<file>** in the current shell instance rather than executing it in a new shell instance

Shell operators

Comparison operators

- **true** always evaluates to true (exit status **0**)
- **false** always evaluates to false (exit status **1**)
- **&&** is a conditional and
- **||** is a conditional or
- **-gt** is a number greater than
- **-lt** is a number less than
- **-eq** is a number equal to

Input/Output

- **<command> < <file>** takes **<file>** as input for **<command>**
- **<command> > <file>** writes the output of **<command>** into **<file>**
- **<command> < <file1> > <file2>** takes **<file1>** as input for **<command>** and writes the output into **<file2>**
- **<command> >> <file>** appends the output of <**command>** to **<file>**
- **<command1> | <command2>** pipes (that is, passes) the output of **<command1>** to **<command2>**
- (Bashism) **<command> &>> <file>** appends both the normal and error output of **<command>** to **<file>**
- **<command> > /dev/null** redirects the output of **<command>** to basically a void, discarding it (can also discard error output by appending **2> /dev/null**)

Variables

Setting and using variables

- **<var>=<value>** initializes a variable named **<var>** with value **<value>** (note: you cannot have any spaces around **=**)
- **$<var>** expands **<var>** to the value that is bound to it (in this case, **<value>**)
- **export <var>** makes **<var>** accessible in child processes

Built-in variables

- **$?** is the exit status of the last command
- **$#** is the number of arguments
- **$@** is all arguments, where each argument is separated by one space
- **$<number>** is the **<number>**'th argument (note: 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

- **`${<var>}`** expands **`<var>`** to the value that is bound to it (preferred to omitting the curly braces since this removes ambiguity in some situations)
- (Bashism) **`${<var>:<lower_index>:<upper_index>}`** gets a substring of **`<var>`** starting from **`<lower_index>`** until, but not including, **`<upper_index>`** (note: **`:<upper_index>`** can be omitted, in which case the substring will include up to the end of **`<var>`**.)
- **`<var>=$(<expression>)`** sets **`<var>`** to the output of **`<expression>`**, which could be a command or another expression that evaluates to a value (ex: **`<stuff>`** is **`echo hello`**, which evaluates to **`hello`**)
- **`<var>=$((<math_expression>))`** sets **`<var>`** to the result of **`<math_expression>`**; you can do arithmetic inside **`$(())`**.

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. ex: **`for i in `ls`; do...`**

Control flow

Evaluating conditional expressions

- **`test <expression>`** returns **`true`** or **`false`** based on the truth of the expression (ex: **`5 -lt 3`**)
- **`[ <expression> ]`** is equivalent to **`test <expression>`** (note: you need a space between **`<expression>`** and the square brackets)
- You can chain **`[ <expression> ]`**'s with **`&&`** and **`||`** operators (ex: **`[ <expression1> ] && [ <expression2> ]`**
- **`! [ <expression> ]`** negates **`[ <expression> ]`** (note the space between **`!`** and **`[`**)
- (Bashism) **`[[ <expression> ]]`** is like **`[ <expression> ]`** but it allows for more string operators, such as **`>`** and **`<`** for string comparison
- (Bashism) **`(( <expression> ))`** is like **`[ <expression> ]`** but it allows for more number operators, such as **`>`** and **`<`** for number comparison

If statements

```
if <test_expression>; then
      <commands>
elif <test_expression2>; then
      <commands2>
else
      <alt_commands>
fi
```

While loops

1.
```
while <test_expression>; do
      <commands>
done
```
2.
```
until <test_expression>; do
      <commands>
done
```

For loops

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

Where **<list>** is a space-delimited sequence of tokens such as **1 2 3** or **$(seq 1 10)**.

Functions (Bashism)

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

Functions can then be called with: **<function-name> <arg1> <arg2>** (remember, you can extract the values of **<arg1>** and **<arg2>** in the function body using **$1** and **$2** respectively)

Case statements

```
case <expression> in
      <value>)
            <commands>
            ;;
      <value2>)
            <commands2>
            ;;
      <value3>)
            <commands3>
            ;;
esac
```

Executables and shell scripts

About shell scripts

Shell scripts contain commands that are run when the script is run.

Before the commands, shell scripts should contain:

- **`#!/bin/bash`**
    - This tells your terminal that when running 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 code 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>` shows the read-write-executable bits for **`<file>`**

`chmod +x <file>` makes **`<file>`** executable

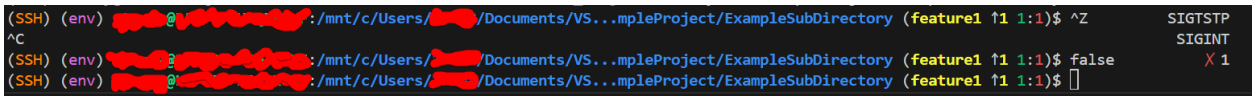
`./<file>` runs **`<file>`** if it's executable

Miscellaneous information

- You can use **ctrl+arrow keys** to move your cursor backward or forward one word in the terminal
- You can use **ctrl+l** to clear the terminal
- You may be able to use the **tab key** to autocomplete a command
- You can use regular expression notation such as **`*`** and **`\w`** for and it will expand to all matches. ex: **`rm *.txt`**
- **`du -h`** lists how much storage different directories use

Shell configuration

I configured my shell to a certain extent. 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.



The GitHub repo for my shell has the full code: 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. It's really strange because in most programming languages you probably know, `:` is a reserved symbol in that language and can't be used when naming a variable. But in Bash, it's no problem.
- `()` denotes that the function does not take in any arguments.
- `{` and `};` enclose the function body.
- `: | :` calls the function `:` 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 is recursive.
- `&` makes the code run in the background.
- The final `:` calls the function `:` outside of the function body, starting the exponential chain.