Semester 1 2024

Astroinformatics I

Tutorial 2: Linux

In this tutorial session, we will work with some basic `bash` commands to help you familiarize yourself with working with the `bash` shell.

1 Which bash?

First you need to find out where is your `bash` interpreter located. Enter the following into your command line:

```
$ which bash
/bin/bash
```

This command reveals that the Bash shell is stored in `/bin/bash`. We will use this information when writing our shell scripts.

2 Our first shell script

For our first shell script, we'll keep the tradition for programming languages to write a script which only outputs `Hello World`.
Create a file named `helloworld.sh` with the following content:

```
##!/bin/bash
# This is a comment!
echo Hello World
# This is a comment, too!
```

Question: What does each line do? Think about it.

Try to run it.
If it wasn't running at your first attempt, you likely forgot to make the file executable.
There are multiple ways for doing so:

```
$ chmod 755 helloworld.sh
```

you could also use:

```
$ chmod +x helloworld.sh
```

After doing that, you can run your script with

```
$ ./helloworld.sh
```

Your screen should then look like this:

```
$ chmod 755 helloworld.sh
$ ./helloworld.sh
Hello World
$
```

Now you are ready to execute your first bash script:

```
$ ./helloworld.sh
```

Question: Which output would you expect to get when adding more spaces, or a tab, between `Hello` and `World`? Try it out.

The output is exactly the same! The reason for this is: We are calling the echo program with two arguments; it doesn't care about the gaps in between them. Now modify the code again:

```
#!/bin/sh
# This is a comment!
echo "Hello      World"       # This is a comment, too!
```

3 Working with variables

We have seen in the lecture that `bash` scripts can contain variables. What is often very helpful is that we can interactively set variable names using the `read` command.
The following script asks you for your name then greets you personally:

```
#!/bin/sh
echo What is your name?
read MY_NAME
echo "Hello $MY_NAME - hope you're well."
```

Task: Try out the above script. Then make a script that does the following: 1. read in a folder path, 2. count the number of files within, 3. output the number of files. It's allowed to search the internet :)

3.1 Variable and file name naming conventions

In `bash` scripting, use the following naming conventions:

- Variable names should start with a letter or an underscore (_).
- Variable names can contain letters, numbers, and underscores.
- Variable names are case-sensitive.
- Variable names should not contain spaces or special characters.
- Use descriptive names that reflect the purpose of the variable.
- Avoid using reserved keywords, such as `if`, `then`, `else`, `fi`, and so on as variable names.

4 Wildcards

Wildcards are likely nothing new for you if you have used Unix at all before.
However, it is not intuitively obvious how they can be used in shell scripts. Here we are trying to predit what the effect of using different syntaxes are. This will be helpful later on to avoid common mistakes and bugs.

Question: Think first how you would copy all the files from `tmp/a` into `tmp/b`. How would you copy only all the `.txt` files?

You might have come up with:

```
$ cp tmp/a/* tmp/b/
$ cp tmp/a/*.txt tmp/b/
```

Make test directories and try it out.

Question: Based on this, how would you list the files in `/tmp/a/` without using `ls /tmp/a/`?

How about `echo /tmp/a/*`? What are the two key differences between this and the ls output? How can this be useful? Or a hinderance? How could you rename all `.txt` files to `.bak`? Note that

```
$ mv *.txt *.bak
```

will not have the desired effect; think about how this gets expanded by the shell before it is passed to `mv`. Try this using echo instead of `mv` if this helps.
Hint: You might need to look up something online, as it uses a few concepts not yet covered.

5 Escape Characters

Certain characters are significant to the `bash` shell. For example, we have seen that the use of double quotes (") characters affect how spaces and TAB characters are treated, for example:

```
$ echo Hello       World
Hello World
$ echo "Hello       World"
Hello     World
```

So how do we display: `Hello "World"`?

```
$ echo "Hello   \"World\""
```

The first and last " characters wrap the text into one parameter which is passed to `echo`, so that the spacing between the two words is kept as is. But the following code:

```
$ echo "Hello   " World ""
```

would be interpreted as three parameters:

```
    "Hello   "
    World
    ""
```

```
Hello    World
```

Note that we lose the quotes entirely. This is because the first and second quotes mark off the Hello and following spaces; the second argument is an unquoted "World" and the third argument is the empty string; "".
Notice that this:

```
$ echo "Hello   "World""
```

is actually only one parameter (no spaces between the quoted parameters), and that you can test this by replacing the `echo` command with (for example) `ls`.
Most characters (`*`, `'`, etc) are not interpreted (i.e., they are taken literally) when we place them in double quotes (`""`). They are taken as is and passed on to the command being called. See the following example using the asterisk (`*`):

```
$ echo *
case.shtml escape.shtml first.shtml
functions.shtml hints.shtml index.shtml
ip-primer.txt raid1+0.txt
$ echo *txt
ip-primer.txt raid1+0.txt
$ echo "*"
*
$ echo "*txt"
*txt
```

In the first example, `*` is expanded to the meaning of "all files in the current directory".
In the second example, `*txt` means "all files ending in `txt`".
In the third example, we put the `*` in double quotes, and it is interpreted literally.
In the fourth example, the same applies, but we have appended `txt` to the string.
However, `"`, `$`, `` ` ``, and `\` are still interpreted by the shell, even if enclosed in double quotes.
The backslash (\) character is used to mark these special characters so that they are not interpreted by the shell, but passed on to the command being run (for example, `echo`). So to output the string: (Assuming that the value of `$X` is `5`):

```
A quote is ", backslash is \, backtick is `.
A few spaces are    and dollar is $. $X is 5.
```

we would have to write:

```
$ echo "A quote is \", backslash is \\, backtick is \`."
A quote is ", backslash is \, backtick is `.
$ echo "A few spaces are    and dollar is \$. \$X is ${X}."
A few spaces are    and dollar is $. $X is 5.
```

We have seen why the `"` is special for preserving spacing. Dollar (`$`) is special because it marks a variable, so `$X` is replaced by the shell with the contents of the variable `X`. Backslash (\) is special because it is itself used to mark other characters off. So backslash itself must be escaped to show that it is to be taken literally.