Shell Tools and Scripting

In this lecture, we will present some of the basics of using bash as a scripting language along with a number of shell tools that cover several of the most common tasks that you will be constantly performing in the command line.

Shell Scripting

So far we have seen how to execute commands in the shell and pipe them together. However, in many scenarios you will want to perform a series of commands and make use of control flow expressions like conditionals or loops.

Shell scripts are the next step in complexity. Most shells have their own scripting language with variables, control flow and its own syntax. What makes shell scripting different from other scripting programming language is that it is optimized for performing shell-related tasks. Thus, creating command pipelines, saving results into files, and reading from standard input are primitives in shell scripting, which makes it easier to use than general purpose scripting languages. For this section we will focus on bash scripting since it is the most common.

To assign variables in bash, use the syntax `foo=bar` and access the value of the variable with `$foo`. Note that `foo = bar` will not work since it is interpreted as calling the `foo` program with arguments `=` and `bar`. In general, in shell scripts the space character will perform argument splitting. This behavior can be confusing to use at first, so always check for that.

Strings in bash can be defined with `'` and `"` delimiters, but they are not equivalent. Strings delimited with `'` are literal strings and will not substitute variable values whereas `"` delimited strings will.

```
foo=bar
echo "$foo"
# prints bar
echo '$foo'
# prints $foo
```

As with most programming languages, bash supports control flow techniques including `if`, `case`, `while` and `for`. Similarly, `bash` has functions that take arguments and can operate with them. Here is an example of a function that creates a directory and `cd`s into it.

```
mcd () {
    mkdir -p "$1"
    cd "$1"
}
```

Here `$1` is the first argument to the script/function. Unlike other scripting languages, bash uses a variety of special variables to refer to arguments, error codes, and other relevant variables. Below is a list of some of them. A more comprehensive list can be found here.

- `$0` - Name of the script
- `$1` to `$9` - Arguments to the script. `$1` is the first argument and so on.
- `$@` - All the arguments
- `$#` - Number of arguments
- `$?` - Return code of the previous command
- `$$` - Process identification number (PID) for the current script
- `!!` - Entire last command, including arguments. A common pattern is to execute a command only for it to fail due to missing permissions; you can quickly re-execute the command with sudo by doing `sudo !!`
- `$_` - Last argument from the last command. If you are in an interactive shell, you can also quickly get this value by typing `Esc` followed by `.` or `Alt+.`

Commands will often return output using `STDOUT`, errors through `STDERR`, and a Return Code to report errors in a more script-friendly manner. The return code or exit status is the way scripts/commands have to communicate how execution went. A value of 0 usually means everything went OK; anything different from 0 means an error occurred.

Exit codes can be used to conditionally execute commands using `&&` (and operator) and `||` (or operator), both of which are short-circuiting operators. Commands can also be separated within the same line using a semicolon `;`. The `true` program will always have a 0 return code and the `false` command will always have a 1 return code. Let's see some examples

```
false || echo "Oops, fail"
# Oops, fail

true || echo "Will not be printed"
#

true && echo "Things went well"
# Things went well

false && echo "Will not be printed"
#

true ; echo "This will always run"
# This will always run

false ; echo "This will always run"
# This will always run
```

Another common pattern is wanting to get the output of a command as a variable. This can be done with *command substitution*. Whenever you place `$( CMD )` it will execute `CMD`, get the output of the command and substitute it in place. For example, if you do `for file in $(ls)`, the shell will first call `ls` and then iterate over those values. A lesser known similar feature is *process substitution*, `<( CMD )` will execute `CMD` and place the output in a temporary file and substitute the `<()` with that file's name. This is useful when commands expect values to be passed by file instead of by STDIN. For example, `diff <(ls foo) <(ls bar)` will show differences between files in dirs `foo` and `bar`.

Since that was a huge information dump, let's see an example that showcases some of these features. It will iterate through the arguments we provide, `grep` for the string `foobar`, and append it to the file as a comment if it's not found.

```bash
#!/bin/bash

echo "Starting program at $(date)" # Date will be substituted

echo "Running program $0 with $# arguments with pid $$"

for file in "$@"; do
    grep foobar "$file" > /dev/null 2> /dev/null
    # When pattern is not found, grep has exit status 1
    # We redirect STDOUT and STDERR to a null register since we do not ca
    if [[ $? -ne 0 ]]; then
        echo "File $file does not have any foobar, adding one"
        echo "# foobar" >> "$file"
    fi
done
```

In the comparison we tested whether `$?` was not equal to 0. Bash implements many comparisons of this sort - you can find a detailed list in the manpage for `test`. When performing comparisons in bash, try to use double brackets `[[ ]]` in favor of simple brackets `[ ]`. Chances of making mistakes are lower although it won't be portable to `sh`. A more detailed explanation can be found here.

When launching scripts, you will often want to provide arguments that are similar. Bash has ways of making this easier, expanding expressions by carrying out filename expansion. These techniques are often referred to as shell *globbing*.

- Wildcards - Whenever you want to perform some sort of wildcard matching, you can use `?` and `*` to match one or any amount of characters respectively. For instance, given files `foo`, `foo1`, `foo2`, `foo10` and `bar`, the command `rm foo?` will delete `foo1` and `foo2` whereas `rm foo*` will delete all but `bar`.
- Curly braces `{}` - Whenever you have a common substring in a series of commands, you can use curly braces for bash to expand this automatically. This comes in very handy when moving or converting files.

```
convert image.{png,jpg}
# Will expand to
convert image.png image.jpg

cp /path/to/project/{foo,bar,baz}.sh /newpath
# Will expand to
cp /path/to/project/foo.sh /path/to/project/bar.sh /path/to/project/baz.s

# Globbing techniques can also be combined
mv *{.py,.sh} folder
# Will move all *.py and *.sh files


mkdir foo bar
# This creates files foo/a, foo/b, ... foo/h, bar/a, bar/b, ... bar/h
touch {foo,bar}/{a..h}
touch foo/x bar/y
# Show differences between files in foo and bar
diff <(ls foo) <(ls bar)
# Outputs
# < x
# ---
# > y
```

Writing `bash` scripts can be tricky and unintuitive. There are tools like shellcheck that will help you find errors in your sh/bash scripts.

Note that scripts need not necessarily be written in bash to be called from the terminal. For instance, here's a simple Python script that outputs its arguments in reversed order:

```
#!/usr/local/bin/python
import sys
for arg in reversed(sys.argv[1:]):
    print(arg)
```

The kernel knows to execute this script with a python interpreter instead of a shell command because we included a shebang line at the top of the script. It is good practice to write shebang lines using the `env` command that will resolve to wherever the command lives in the system, increasing the portability of your scripts. To resolve the location, `env` will make use of the `PATH` environment variable we introduced in the first lecture. For this example the shebang line would look like `#!/usr/bin/env python`.

Some differences between shell functions and scripts that you should keep in mind are:

- Functions have to be in the same language as the shell, while scripts can be written in any language. This is why including a shebang for scripts is important.
- Functions are loaded once when their definition is read. Scripts are loaded every time they are executed. This makes functions slightly faster to load, but whenever you change them you will have to reload their definition.
- Functions are executed in the current shell environment whereas scripts execute in their own process. Thus, functions can modify environment variables, e.g. change your current directory, whereas scripts can't. Scripts will be passed by value environment variables that have been exported using `export`
- As with any programming language, functions are a powerful construct to achieve modularity, code reuse, and clarity of shell code. Often shell scripts will include their own function definitions.

Shell Tools

Finding how to use commands

At this point, you might be wondering how to find the flags for the commands in the aliasing section such as `ls -l`, `mv -i` and `mkdir -p`. More generally, given a command, how do you go about finding out what it does and its different options? You could always start googling, but since UNIX predates StackOverflow, there are built-in ways of getting this information.

As we saw in the shell lecture, the first-order approach is to call said command with the `-h` or `--help` flags. A more detailed approach is to use the `man` command. Short for manual, `man` provides a manual page (called manpage) for a command you specify. For example, `man rm` will output the behavior of the `rm` command along with the flags that it takes, including the `-i` flag we showed earlier. In fact, what I have been linking so far for every command is the online version of the Linux manpages for the commands. Even non-native commands that you install will have manpage entries if the developer wrote them and included them as part of the installation process. For interactive tools such as the ones based on ncurses, help for the commands can often be accessed within the program using the `:help` command or typing `?`.

Sometimes manpages can provide overly detailed descriptions of the commands, making it hard to decipher what flags/syntax to use for common use cases. TLDR pages are a nifty complementary solution that focuses on giving example use cases of a command so you can quickly figure out which options to use. For instance, I find myself referring back to the tldr pages for `tar` and `ffmpeg` way more often than the manpages.

Finding files

One of the most common repetitive tasks that every programmer faces is finding files or directories. All UNIX-like systems come packaged with `find`, a great shell tool to find files. `find` will recursively search for files matching some criteria. Some examples:

```
# Find all directories named src
find . -name src -type d
# Find all python files that have a folder named test in their path
find . -path '*/test/*.py' -type f
# Find all files modified in the last day
find . -mtime -1
# Find all zip files with size in range 500k to 10M
find . -size +500k -size -10M -name '*.tar.gz'
```

Beyond listing files, find can also perform actions over files that match your query. This property can be incredibly helpful to simplify what could be fairly monotonous tasks.

```
# Delete all files with .tmp extension
find . -name '*.tmp' -exec rm {} \;
# Find all PNG files and convert them to JPG
find . -name '*.png' -exec convert {} {}.jpg \;
```

Despite `find`'s ubiquitousness, its syntax can sometimes be tricky to remember. For instance, to simply find files that match some pattern `PATTERN` you have to execute `find -name '*PATTERN*'` (or `-iname` if you want the pattern matching to be case insensitive). You could start building aliases for those scenarios, but part of the shell philosophy is that it is good to explore alternatives. Remember, one of the best properties of the shell is that you are just calling programs, so you can find (or even write yourself) replacements for some. For instance, `fd` is a simple, fast, and user-friendly alternative to `find`. It offers some nice defaults like colorized output, default regex matching, and Unicode support. It also has, in my opinion, a more intuitive syntax. For example, the syntax to find a pattern `PATTERN` is `fd PATTERN`.

Most would agree that `find` and `fd` are good, but some of you might be wondering about the efficiency of looking for files every time versus compiling some sort of index or database for quickly searching. That is what `locate` is for. `locate` uses a database that is updated using `updatedb`. In most systems, `updatedb` is updated daily via `cron`. Therefore one trade-off between the two is speed vs freshness. Moreover `find` and similar tools can also find files using attributes such as file size, modification time, or file permissions, while `locate` just uses the file name. A more in-depth comparison can be found here.

Finding code

Finding files by name is useful, but quite often you want to search based on file *content*. A common scenario is wanting to search for all files that contain some pattern, along with where in those files said pattern occurs. To achieve this, most UNIX-like systems provide `grep`, a generic tool for matching patterns from the input text. `grep` is an incredibly valuable shell tool that we will cover in greater detail during the data wrangling lecture.

For now, know that `grep` has many flags that make it a very versatile tool. Some I frequently use are `-C` for getting **C**ontext around the matching line and `-v` for in**v**erting the match, i.e. print all lines that do **not** match the pattern. For example, `grep -C 5` will print 5 lines before and after the match. When it comes to quickly searching through many files, you want to use `-R` since it will **R**ecursively go into directories and look for files for the matching string.

But `grep -R` can be improved in many ways, such as ignoring `.git` folders, using multi CPU support, &c. Many `grep` alternatives have been developed, including ack, ag and rg. All of them are fantastic and pretty much provide the same functionality. For now I am sticking with ripgrep (`rg`), given how fast and intuitive it is. Some examples:

```
# Find all python files where I used the requests library
rg -t py 'import requests'
# Find all files (including hidden files) without a shebang line
rg -u --files-without-match "^#!"
# Find all matches of foo and print the following 5 lines
rg foo -A 5
# Print statistics of matches (# of matched lines and files )
rg --stats PATTERN
```

Note that as with `find`/`fd`, it is important that you know that these problems can be quickly solved using one of these tools, while the specific tools you use are not as important.

Finding shell commands

So far we have seen how to find files and code, but as you start spending more time in the shell, you may want to find specific commands you typed at some point. The first thing to know is that typing the up arrow will give you back your last command, and if you keep pressing it you will slowly go through your shell history.

The `history` command will let you access your shell history programmatically. It will print your shell history to the standard output. If we want to search there we can pipe that output to `grep` and search for patterns. `history | grep find` will print commands that contain the substring "find".

In most shells, you can make use of `Ctrl+R` to perform backwards search through your history. After pressing `Ctrl+R`, you can type a substring you want to match for commands in your history. As you keep pressing it, you will cycle through the matches in your history. This can also be enabled with the UP/DOWN arrows in zsh. A nice addition on top of `Ctrl+R` comes with using fzf bindings. `fzf` is a general-purpose fuzzy finder that can be used with many commands. Here it is used to fuzzily match through your history and present results in a convenient and visually pleasing manner.

Another cool history-related trick I really enjoy is **history-based autosuggestions**. First introduced by the fish shell, this feature dynamically autocompletes your current shell command with the most recent command that you typed that shares a common prefix with it. It can be enabled in zsh and it is a great quality of life trick for your shell.

You can modify your shell's history behavior, like preventing commands with a leading space from being included. This comes in handy when you are typing commands with passwords or other bits of sensitive information. To do this, add `HISTCONTROL=ignorespace` to your `.bashrc` or `setopt HIST_IGNORE_SPACE` to your `.zshrc`. If you make the mistake of not adding the leading space, you can always manually remove the entry by editing your `.bash_history` or `.zhistory`.

Directory Navigation

So far, we have assumed that you are already where you need to be to perform these actions. But how do you go about quickly navigating directories? There are many simple ways that you could do this, such as writing shell aliases or creating symlinks with `ln -s`, but the truth is that developers have figured out quite clever and sophisticated solutions by now.

As with the theme of this course, you often want to optimize for the common case. Finding frequent and/or recent files and directories can be done through tools like `fasd` and `autojump`. Fasd ranks files and directories by *frecency*, that is, by both *frequency* and *recency*. By default, `fasd` adds a `z` command that you can use to quickly `cd` using a substring of a *frecent* directory. For example, if you often go to `/home/user/files/cool_project` you can simply use `z cool` to jump there. Using autojump, this same change of directory could be accomplished using `j cool`.

More complex tools exist to quickly get an overview of a directory structure: `tree`, `broot` or even full fledged file managers like `nnn` or `ranger`.

Exercises

1. Read `man ls` and write an `ls` command that lists files in the following manner
   - Includes all files, including hidden files
   - Sizes are listed in human readable format (e.g. 454M instead of 454279954)
   - Files are ordered by recency
   - Output is colorized

   A sample output would look like this

```
-rw-r--r--   1 user group 1.1M Jan 14 09:53 baz
drwxr-xr-x   5 user group  160 Jan 14 09:53 .
-rw-r--r--   1 user group  514 Jan 14 06:42 bar
-rw-r--r--   1 user group 106M Jan 13 12:12 foo
drwx------+ 47 user group 1.5K Jan 12 18:08 ..
```

2. Write bash functions `marco` and `polo` that do the following. Whenever you execute `marco` the current working directory should be saved in some manner, then when you execute `polo`, no matter what directory you are in, `polo` should `cd` you back to the directory where you executed `marco`. For ease of debugging you can write the code in a file `marco.sh` and (re)load the definitions to your shell by executing `source marco.sh`.

3. Say you have a command that fails rarely. In order to debug it you need to capture its output but it can be time consuming to get a failure run. Write a bash script that runs the following script until it fails and captures its standard output and error streams to files and prints everything at the end. Bonus points if you can also report how many runs it took for the script to fail.

```bash
#!/usr/bin/env bash

n=$(( RANDOM % 100 ))

if [[ n -eq 42 ]]; then
   echo "Something went wrong"
   >&2 echo "The error was using magic numbers"
   exit 1
fi

echo "Everything went according to plan"
```

4. As we covered in the lecture `find`'s `-exec` can be very powerful for performing operations over the files we are searching for. However, what if we want to do something with **all** the files, like creating a zip file? As you have seen so far commands will take input from both arguments and STDIN. When piping commands, we are connecting STDOUT to STDIN, but some commands like `tar` take inputs from arguments. To bridge this disconnect there's the `xargs` command which will execute a command using STDIN as arguments. For example `ls | xargs rm` will delete the files in the current directory.

   Your task is to write a command that recursively finds all HTML files in the folder and makes a zip with them. Note that your command should work even if the files have spaces (hint: check `-d` flag for `xargs`).

If you're on macOS, note that the default BSD `find` is different from the one included in GNU coreutils. You can use `-print0` on `find` and the `-0` flag on `xargs`. As a macOS user, you should be aware that command-line utilities shipped with macOS may differ from the GNU counterparts; you can install the GNU versions if you like by using brew.

5. (Advanced) Write a command or script to recursively find the most recently modified file in a directory. More generally, can you list all files by recency?