Lecture 2:

Shell Tools and Scripting

Shell Scripting

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 is optimized for performing shell related tasks. Thus, creating command pipelines, saving results into files or reading from standard input are primitives in shell scripting which makes it easier to use than general purpose scripting languages. This section 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 and it 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 cds 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 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, then you can quickly execute it 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 .

Commands will often return output using STDOUT, errors through STDERR and a Return Code to report errors in a more script friendly manner. Return code or exit status is the way scripts/commands have to communicate how execution went. A value 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). 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.

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"
#

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 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.

```
#!/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 care
about them
    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 favour 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 covering files.

Writing bash scripts can be tricky and unintuitive. There are tools like shellcheck that will help you find

```
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.sh
/newpath

# 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
# ---
```

out 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 shell 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. Often shell scripts will include their own function definitions.

Shell Tools

Finding how to use commands

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 builtin 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 behaviour 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 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 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 be overly detailed descriptions of the commands and it can become hard to decipher what flags/syntax to use for common use cases. TLDR pages are a nifty complementary solution that focuses on giving example uses 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 packed with find, a great shell tool to find files. 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 as part 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 simple, fast and user-friendly alternative to find. It offers some nice defaults like colorized output, default regex matching, Unicode support and it has in what my opinion is a more intuitive syntax. The syntax to find a pattern PATTERN in 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 updating 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 name. A more in depth comparison can be found here.

Finding code

Finding files is useful but quite often you are after what is in the file. 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. It is an incredibly valuable shell tool and we will cover it more in detail during the data wrangling lecture.

grep has many flags that make it a very versatile tool. Some I frequently use are -C for getting Context around the matching line and -v for inverting 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 parsing through many files, you want to use -R since it will Recursively go into directories and look for text files for the matching string.

But grep -R can be improved in many ways, such as ignoring .git folders, using multi CPU support, &c. So there has been no shortage of alternatives developed, including ack, ag and rg.