Shell Scripting

Up to this point, we've covered how to run commands in the shell and connect them using pipes. However, in many cases, you'll want to execute a sequence of commands and utilize control flow statements like conditionals or loops.

Shell scripts represent a more complex step. Most shells include their own scripting languages featuring variables, control flow, and unique syntax. What sets shell scripting apart from other programming languages is its optimization for tasks related to the shell. This makes creating command pipelines, saving outputs to files, and reading from standard input foundational elements in shell scripting, making it simpler to use compared to general-purpose languages. This section will concentrate on bash scripting, as it is the most widely used.

To assign a variable in bash, use the syntax `foo=bar`, and to access the variable's value, use `$foo`. Be aware that `foo = bar` will not work, as it will be interpreted as executing the `foo` program with arguments `=` and `bar`. Generally, spaces in shell scripts lead to argument splitting, which can be confusing at first, so it's important to keep this in mind.

In bash, strings can be defined using single (`'`) or double (`"`) quotes, but they behave differently. Strings in single quotes are literal and do not substitute variable values, while double-quoted strings do.

```bash
foo=bar
echo "$foo"   # outputs bar
echo '$foo'   # outputs $foo
```

Like most programming languages, bash supports control flow structures such as `if`, `case`, `while`, and `for`. Additionally, bash allows for function definitions that can take arguments. Here's an example of a function that creates a directory and navigates into it:

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

```

Here, `$1` represents the first argument passed to the script or function. Unlike other scripting languages, bash utilizes special variables to refer to arguments, error codes, and more. Below is a list of some key variables, with a more detailed list available in the documentation.

- `$0`: Name of the script

- `$1` to `$9`: Arguments to the script, with `$1` being the first argument

- `$@`: All arguments

- `$#`: Number of arguments

- `$?`: Exit status of the last command

- `$$`: Process ID of the current script

- `!!`: The last command including its arguments

- `$_`: Last argument from the last command

Commands typically output results through STDOUT, errors through STDERR, and return codes to communicate execution status. A return code of 0 usually indicates success, while any other value signifies an error.

You can use exit codes to conditionally execute commands with `&&` (and) and `||` (or), both of which are short-circuiting operators. Commands can also be separated on the same line with a semicolon `;`. The true command will always return 0, while the false command returns 1. Here are some examples:

```bash

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

true || echo "Will not be printed"  # No output

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

false && echo "Will not be printed"  # No output

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

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

```

Another common scenario is capturing the output of a command into a variable. This is accomplished using command substitution. By placing `$(CMD)`, the shell executes `CMD`, captures its output, and substitutes it in place. For instance, `for file in $(ls)` will first execute `ls` and then loop through its output. A similar feature, process substitution, uses `<(CMD)` to execute `CMD`, placing its output in a temporary file and substituting `<()` with that filename. This is useful when commands require input from a file rather than STDIN, such as in `diff <(ls foo) <(ls bar)` to compare files in directories `foo` and `bar`.

Given the extensive information provided, let's look at an example that demonstrates some of these features. It will iterate over provided arguments, check for the string "foobar", and append it to the file as a comment if it's absent.

```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 the pattern is not found, grep returns an exit status of 1
   # STDOUT and STDERR are redirected to /dev/null since we don't need to see them
   if [[ $? -ne 0 ]]; then
      echo "File $file does not contain foobar, adding one"
      echo "# foobar" >> "$file"
   fi
done
```
```

In this case, we checked if `$?` was not equal to 0. Bash supports many comparisons like this—details can be found in the `man` page for `test`. When making comparisons, prefer using double brackets `[[ ]]` instead of single brackets `[ ]` to reduce mistakes, though this may affect portability to `sh`. More information can be found in the documentation.

When running scripts, you often want to provide similar arguments. Bash facilitates this with filename expansion techniques, commonly known as shell globbing.

- Wildcards: Use `?` and `*` for wildcard matching, where `?` matches one character and `*` matches any number of characters. For example, `rm foo?` will remove `foo1` and `foo2`, while `rm foo*` will delete everything except `bar`.

- Curly braces `{}`: If you have a common substring across several commands, you can use curly braces to expand this automatically. This is handy for moving or converting files.

```bash
convert image.{png,jpg}  # Expands to convert image.png image.jpg


cp /path/to/project/{foo,bar,baz}.sh /newpath  # Expands to multiple cp commands
```

You can also combine globbing techniques:

```bash
mv *{.py,.sh} folder  # Moves all .py and .sh files

mkdir foo bar  # Creates foo/a, foo/b, etc.

touch {foo,bar}/{a..h}  # Creates multiple files
```

Writing bash scripts can be complex and unintuitive. Tools like `shellcheck` can help identify errors in your bash scripts.

Scripts don’t necessarily need to be written in bash to be executed from the terminal. For example, here’s a simple Python script that prints its arguments in reverse order:

```python
#!/usr/local/bin/python
```

```
import sys

for arg in reversed(sys.argv[1:]):
    print(arg)
```

The kernel knows to run this script with a Python interpreter due to the shebang line at the top. It's advisable to use a shebang line with the `env` command to enhance portability. For this example, it would be `#!/usr/bin/env python`.

Here are some distinctions between shell functions and scripts:

- Functions must be in the same language as the shell, while scripts can be in any language, necessitating the shebang for scripts.

- Functions are loaded once when their definitions are read, whereas scripts are loaded each time they are executed. This makes functions slightly faster to load, but they require reloading upon changes.

- Functions execute within the current shell environment, allowing them to modify environment variables, while scripts run in their own process and can only receive exported environment variables.

As in any programming language, functions are powerful tools for achieving modularity, code reuse, and clarity in shell code. Shell scripts often include function definitions.

Shell Tools

Finding Command Usage

At this point, you might be curious about how to discover the flags for commands like `ls -l`, `mv -i`, and `mkdir -p`. More generally, how do you learn about a command and its options? While Googling is an option, UNIX provides built-in methods to access this information.

The first approach is to use the `-h` or `--help` flags with the command. A more comprehensive method is to utilize the `man` command, which provides manual pages for specified commands. For example, `man rm` will display details about the `rm` command, including its available flags. Additionally, most installed commands will have corresponding manpage entries if provided by the developer. Interactive tools often allow access to command help via `:help` or `?`.

Sometimes, manpages can be overly detailed, making it hard to find specific flags or syntax for common tasks. TLDR pages offer a handy alternative, focusing on practical examples of commands to help quickly determine which options to use. For instance, I often refer to TLDR pages for commands like `tar` and `ffmpeg` more frequently than manpages.

Finding Files

A common task for any programmer is locating files or directories. All UNIX-like systems come with the `find` command, a powerful tool for searching files based on specified criteria. Here are some examples:

```bash

# 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 between 500k and


10M

find . -size +500k -size -10M -name '*.tar.gz'

```

Beyond just listing files, `find` can perform actions on matching files, streamlining repetitive tasks:

```bash

# Delete all .tmp files

find . -name '*.tmp' -exec rm {} \;

# Convert all PNG files to JPG
```

find . -name '*.png' -exec convert {}{}.jpg \;

```

Though `find` is widely used, its syntax can be tricky to remember. For instance, to find files matching a pattern, you need to execute `find -name '*PATTERN*'` (or `-iname` for case-insensitivity). Instead of creating aliases, remember that the shell is designed for program calls, so you can find or create alternatives. For instance, `fd` is a fast, user-friendly alternative to `find`, offering useful defaults like colorized output and regex matching. Its syntax is often more intuitive, e.g., `fd PATTERN`.

While `find` and `fd` are solid options, some might wonder about the efficiency of searching for files repeatedly versus maintaining an index for quick searches. This is where `locate` comes in, using a database updated by `updatedb`, typically on a daily schedule. The trade-off here is speed versus freshness; while `find` can search by attributes (like file size or modification time), `locate` only uses filenames.

**Finding Code**

Searching files by name is useful, but often you need to find files based on their content. A common scenario is searching for files containing a specific pattern and identifying where that pattern occurs. Most UNIX-like systems provide `grep`, a versatile tool for matching patterns in text, which we will explore in greater depth later.

For now, know that `grep` has many flags making it highly adaptable. Some frequently used options include `-C` for displaying context around matches and `-v` for inverting matches (showing lines that do not match the pattern). For example, `grep -C 5` shows five lines before and after a match. To search through many files, use `-R` for recursive searches.

However, `grep -R` can be enhanced to ignore `.git` directories, support multiple CPU cores, etc. Various alternatives to `grep` exist, including `ack`, `ag`, and `rg`, all of which provide similar functionality. Currently, I prefer `ripgrep` (rg) due to its speed and intuitiveness. Some examples include:

```bash

# Find all Python files using the requests library

rg -t py 'import requests'

# Find files without a shebang line

rg -u --files-without-match "^#\!"

# Find all matches for foo and print the following 5 lines

rg foo -A 5

# Display statistics about matches

rg --stats PATTERN

```

As with `find` and `fd`, it's essential to recognize that the specific tools are less important than knowing how to solve these problems efficiently.

Finding Shell Commands

We've looked at how to find files and code, but you may also want to locate specific commands you've previously entered in the shell. The simplest way is by pressing the up arrow to retrieve your last command, and you can continue pressing it to cycle through your command history.

The `history` command allows you to view your shell history programmatically, displaying it in the standard output. To search this history, you can pipe the output to `grep`, for instance, `history | grep find` to find commands that include "find".

In most shells, you can use `Ctrl+R` for a reverse search through your history. After pressing `Ctrl+R`, type a substring to match against commands in your history. Continuing to press it will cycle through the matches. This functionality can also be enabled with UP/DOWN arrows in `zsh`. Adding `fzf` bindings enhances this, allowing fuzzy matching through your history with visually appealing results.

Another useful feature is history-based autosuggestions, first introduced in the fish shell. This feature dynamically autocompletes your current command with the most recent command sharing a common prefix. It can be enabled in `zsh` and significantly enhances the user experience in the shell.

You can adjust your shell's history settings, such as preventing commands starting with a space from being recorded. This is helpful for sensitive commands. To do this, add `HISTCONTROL=ignorespace` to your `.bashrc` or `setopt HIST_IGNORE_SPACE` to your `.zshrc`. If you forget to use the leading space, you can manually remove the entry from your history file.
```

Directory Navigation

Thus far, we've assumed you're already in the correct directory to execute these actions. But how can you navigate directories quickly? There are simple methods like using shell aliases or creating symbolic links with `ln -s`, but developers have created more clever solutions.

As with the course's theme, it's beneficial to optimize for common tasks. Tools like `fasd` and `autojump` help find frequently or recently accessed files and directories. `fasd` ranks files and directories by both frequency and recency, allowing you to use the command `z` followed by a substring to quickly change directories. For example, `z cool` can take you to `/home/user/files/cool_project`. Similarly, `autojump` allows for directory navigation using `j cool`.