Shell Tools and Scripting

Introduction to Shell Scripting

- **Bash scripting basics**
  - Execution of commands
  - Piping them together
  - Control flow expressions: conditionals, loops

Introduction to Shell Scripting (Cont'd)

- **Advantages**
  - Optimized for shell-related tasks
  - Easier for command pipelines, file operations, and standard input
- **Focus**
  - Bash scripting (most common shell scripting language)

Variables in Bash

- **Assignment**
  - Correct: `foo=bar`
  - Incorrect: `foo = bar` (calls `foo` with args `=` and `bar`)
- **String Delimiters**
  - `'` for literal strings (no variable substitution)
  - `"` for strings with variable substitution

Variables in Bash (Cont'd)

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

Control Flow in Bash

- **Supported structures**
  - `if`
  - `case`
  - `while`
  - `for`
- **Functions**
  - Can take arguments
  - Example: Function to create a directory and `cd` into it

Control Flow in Bash (Cont'd)

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

- `$1` refers to the first argument of the script/function

Special Variables

- `$0` - Name of the script
- `$1` to `$9` - Arguments to the script
- `$@` - All the arguments
- `$#` - Number of arguments
- `$?` - Return code of the last command
- `$$` - PID for the current script
- `!!` - Entire last command (useful with `sudo !!`)
- `$_` - Last argument from the last command

More details here.

Command Execution and Return Codes

- **STDOUT** and **STDERR**
- **Return Code** (Exit Status)
  - `0` for success
  - Non-zero for error
- **Conditional Execution**
  - `&&` and `||` (short-circuiting operators)
  - Semicolon `;` to separate commands

Command Execution and Return Codes (Cont'd)

```
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
```

Command and Process Substitution

- **Command Substitution**: `$( CMD )`
  - Executes `CMD`, substitutes output
- **Process Substitution**: `<( CMD )`
  - Executes `CMD`, passes output as a temporary file

Command and Process Substitution (Cont'd)

```
# Command Substitution Example
for file in $(ls); do
  echo $file
done

# Process Substitution Example
diff <(ls foo) <(ls bar) # Shows differences between files in dirs foo and bar
```

Example Bash Script

- Iterates through arguments
- Searches for string `foobar`
- Appends it to the file as a comment if not found

Example Bash Script (Cont'd)

```
#!/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
    if [[ $? -ne 0 ]]; then
        echo "File $file does not have any foobar, adding one"
        echo "# foobar" >> "$file"
    fi
done
```

Example Bash Script (Cont'd) - `grep` Command

- `$(date)` - Command substitution
- `$0`, `$#`, `$?` - Special variables
- `grep` - Command usage with redirection

Comparisons and Globbing in Bash

- **Comparisons**
  - Use `[[ ]]` for conditional expressions
  - More robust and less error-prone than `[ ]`
- **Globbing Techniques**
  - Wildcards (`?` and `*`)
  - Curly braces (`{}`) for common substrings

Comparisons and Globbing in Bash (Cont'd)

```
# Wildcard examples
rm foo?  # Removes foo1, foo2, etc.
rm foo*  # Removes all foo* files

# Curly braces example
convert image.{png,jpg} # Converts image.png to image.jpg
```

- Refer to the `test` manpage for details
  - test manpage

More on Globbing and Shellcheck

- **Curly Braces Expansion**

```
cp /path/to/project/{foo,bar,baz}.sh /newpath
# Expands to separate cp commands for each .sh file
```

- **Combining Globbing Techniques**

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

More on Globbing and Shellcheck (Cont'd)

- **Creating and Comparing Directories**

```
mkdir foo bar
touch {foo,bar}/{a..h}
touch foo/x bar/y
diff <(ls foo) <(ls bar)
# Compares contents of directories foo and bar
```

- **Shellcheck**
  - A tool to find errors in sh/bash scripts
  - Helpful for writing robust scripts
  - Shellcheck on GitHub

Scripting Beyond Bash

- **Non-bash scripts in the terminal**
  - Shebang line determines the interpreter
  - Example: Python script to reverse arguments

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

- **Shebang best practices**
  - Use the `env` command for portability
  - Example shebang: `#!/usr/bin/env python`

Shell Functions vs. Scripts

- **Language**
  - Functions: Same as shell
  - Scripts: Any language (with shebang)
- **Loading**
  - Functions: Loaded once (faster, need to reload after changes)
  - Scripts: Loaded each execution

Shell Functions vs. Scripts (Cont'd)

- **Execution Environment**
  - Functions: Current shell environment (can modify it)
  - Scripts: Own process (can't modify the invoking shell's environment)
- **Use Cases**
  - Functions: Modularity, code reuse, clarity within shell scripts

Shell Tools - Command Information

- **Built-in help**
  - `-h` or `--help` flags
  - Example: `ls --help`
- **Manual pages**
  - `man` command
  - Example: `man rm`

Shell Tools - Command Information (Cont'd)

- **Online Linux manpages**
  - For all linked commands
- **Interactive tools**
  - `:help` command or `?` within the program

Finding Files with `find`, `fd`, and `locate`

`find`

- Recursively search for files matching criteria.
- Examples:
  - Find directories named `src`: `find . -name src -type d`
  - Find python files in `test` folders: `find . -path '*/test/*.py' -type f`
  - Find files modified in the last day: `find . -mtime -1`
  - Find zip files between 500k and 10M: `find . -size +500k -size -10M -name '*.tar.gz'`

Finding Files with `find`, `fd`, and `locate` (Cont'd)

Performing Actions with `find`

- Delete `.tmp` files: `find . -name '*.tmp' -exec rm {} \;`
- Convert PNG to JPG: `find . -name '*.png' -exec convert {} {}.jpg \;`

Alternatives: `fd` and `locate`

- `fd`: Simpler syntax, colorized output, regex matching.
- `locate`: Uses a database for quick searches, trades off freshness for speed.

Finding Code with `grep`, `ack`, `ag`, and `rg`

`grep`

- Search for patterns in files.
- Examples:
  - Get context around matches: `grep -C 5 PATTERN`
  - Invert match: `grep -v PATTERN`
  - Recursive search: `grep -R PATTERN`

Finding Code with `grep`, `ack`, `ag`, and `rg` (Cont'd)

Alternatives: `ack`, `ag`, `rg`

- Faster and with better defaults.
- Examples with `rg` (ripgrep):
  - Find python files using `requests`: `rg -t py 'import requests'`
  - Find files without shebang: `rg -u --files-without-match "^#\!"`
  - Print 5 lines after match: `rg foo -A 5`
  - Print match statistics: `rg --stats PATTERN`

Finding Shell Commands

History and Search

- `history` : Access shell history.
- `Ctrl+R` : Backwards search through history.
- `history | grep find` : Search history for "find".

Fuzzy Finder with `fzf`

- Fuzzy match through history.
- Visually pleasing and convenient.

Finding Shell Commands (Cont'd)

History-based Autosuggestions

- Dynamic autocompletion for shell commands.
- Inspired by `fish`, available in `zsh`.

History Behavior Customization

- Exclude commands with leading space: `HISTCONTROL=ignorespace` or `setopt HIST_IGNORE_SPACE`.
- Manually edit history files: `.bash_history` or `.zsh_history`.

Directory Navigation Tools

Quick Navigation

- `fasd` and `autojump` for frequent/recent directories.
- `fasd` uses *frecency* (frequency + recency) to rank directories.
- `z` and `j` commands for quick directory jumping.

Overview and Management

- `tree`, `broot` for directory structure overview.
- File managers: `nnn`, `ranger` for in-depth directory management.

Exercises

Exercise 1: Enhanced `ls` Command

- List all files (`-a` or `-A` for no `.` and `..`).
- Human-readable sizes (`-h`).
- Order by recency (`-lt` for modification time, newest first).
- Colorized output (`--color=auto`).

Command: `ls -lah --color=auto`

Exercise 2: `marco` and `polo` Functions

```
#!/bin/bash

## marco: Save current directory
marco() {
    export MARCO_DIR=$(pwd)
}

## polo: Go to saved directory
polo() {
    cd "$MARCO_DIR" || return
}
```

Exercise 3: Debugging a Rare Failure

```
#!/bin/bash

count=0
while ./rare_failure_script.sh; do
    ((count++))
done

echo "The script failed after $count runs."
```

- Capture output: `./rare_failure_script.sh >stdout.txt 2>stderr.txt`

Exercise 4: Zip HTML Files

- Find HTML files: `find . -name '*.html'`
- Use `xargs` to handle spaces: `find . -name '*.html' -print0 | xargs -0 zip html_files.zip`

Command: `find . -name '*.html' -print0 | xargs -0 zip html_files.zip`

Exercise 5: Find Most Recently Modified File

```
find . -type f -exec stat --format '%Y :%y %n' {} \; | sort -nr | head -1
```

- List all files by recency:

```
find . -type f -exec stat --format '%Y :%y %n' {} \; | sort -nr
```

- `%Y` gives time of last modification, seconds since Epoch.
- `:%y` gives human-readable modification time.
- `%n` gives file name.
- `sort -nr` sorts numerically and reverses the order.

Note: `stat` syntax may vary between systems.