FLEXION®
VALUE FORWARD

February, 2023

Bash

Programming Languages Guild

Stuff I plan to talk about

- Separating Commands
  - Conditional execution with boolean logic
- Variables
  - Naming
  - Declaring
  - Built-ins
  - Arrays
  - iterating
- Conditionals
  - Simple
  - Compound
  - test operators
  - Regular Expression Matching
- Variable Expansion
  - Nulls
  - Substrings
  - Filename manipulation
  - Pattern search / replace
- Functions
  - Declaring
  - Positional arguments
  - Environmental arguments
  - Named arguments
  - Result codes
  - Returning flat strings
  - Returning Complex Values with jq
- Command Line Parameters
  - Positional arguments
  - Using getopts with short flags
  - Using getopts with long flags
- Better Bash Code
  - Refactor Code using Libraries
  - Unit testing with BATS
  - Documentation with Doxygen
  - Static code analysis with Shellcheck
  - Reformatting with shfmt
  - Safer Bash Tips

Separating Commands

- Semicolons (;) may be used to separate commands
- Boolean AND (&&) if the first command succeeds, do the second command
- Boolean OR (||) if the first command fails, do the second command

```
cp a.txt b.txt ; cat ~/b.txt                    # copy a file and then output the copy

grep -qi "password" ~/test.txt && echo "oops"   # if password is in file, write oops

cd ~/tmp || exit 1                              # if we couldn't cd to a file, quit
```

Variable Names

- Begin with '$' optionally with '{ curly braces }' to disambiguate, case-sensitive

```
$iAmAVariable

$i_am_a_variable

${im_a_variable}

${library__variable}
```

Variables Declaration

- Use 'declare' to create a variable
- May use equal sign (=) to assign an initial value (use care not to mask return values)
- Note: no $ sigil

```
declare foo                # create a new string variable named foo with no value

declare bazzle="blarg"     # create a new variable named bazzle with a value of blarg

declare -i bar=3           # create a new integer variable named bar with a value of 3
```

Built-in Variables

HOME	User's home directory		PPID	Parent's PID
PATH	Directory search path		RANDOM	A random number
PS1	User's prompt		SECONDS	How long shell running
BASH_VERSION	Bash version		EPOCHSECONDS	Current time (epoch)
COLUMNS	Width of the terminal		$*	Positional arguments*
HOSTNAME	Current host's name		$@	Positional arguments*
UID	User ID		$?	Exit status last process
EUID	Effective User ID		$0	Shell / script name
PWD	Current working directory		$$	Current PID

Indexed and Associative Arrays

- Declare with -a or -A

```
declare -a foo
foo=(bar bazzle)
foo[0]=bar ; foo[1]=bazzle

declare -A bell
declare -A bell=([color]=green [sound]=ding)
bell[color]=green ; bell[sound]=ding

printf "The bell sounds like %s" "${bell[sound]}"
```

Iterating Across Indexed and Associative Arrays

- Keys: `${!array[@]}`
- Values: `${array[@]}`

```
for key in "${!array[@]}" ; do printf "key = '%s'\n" "$key" ; done

for value in "${array[@]}" ; do printf "value = '%s'\n" "$value" ; done

for key in "${!array[@]}" ; do printf "%s : %s\n" "$key" "${array[$key]}" ; done
```

Simple Conditionals

- Basic if-then-else structures; elif and else are entirely optional
- Often used with [ ] or the test command
- Semicolons (;) required before then

```
if condition ; then
  action ;
[ elif condition ; then
  actions ; ]
[ else actions ; ]
fi


if condition ; then action ; elif condition ; then action ; else action ; fi
```

Compound Conditionals

- Negate expression: !
- Boolean AND: &&
- Boolean OR: ||
- Override precedence of operators: ( )

```
if [ "$a" = "$b" ] && [ "$b" == "$c" ] ; then echo "a = c" ; fi


if [ "$a" = "$b" ] \
&& [ "$b" = "$c" ] ; then
  echo "a is equal to c"
fi
```

Conditions with test and [ ]

= (POSIX) or == / !=	String equality		-s *file*	File is non-zero size
< or >	Lexicographical comparison [ ] use ASCII while [[ ]] use locale		-r *file*	File is readable
-eq or -ne	Numeric equality		-w *file*	File is writeable
-lt -le -ge -gt	Numeric comparison		-x *file*	File is executable
-v *variable*	Variable has been declared		-f *file*	File is a regular file
-n *string*	String is not empty		-L *file*	File is a symlink
-z *string*	String is empty		-d *file*	File is a directory
-a or -e *file*	File exists		*file1* -nt *file2*	file1 is newer than file2

Conditions with test and [ ]

```
declare foo=bar
declare -i bazzle=3

if [ "$foo" == "bar" ] ; then echo "true" ; else echo "false" ; fi

if test $bazzle > 2 ; then echo "true" ; else echo "false" ; fi

if [ "ant" < "aardvark" ] ; then echo "true" ; else echo "false" ; fi

if [ -f /var/run/docker.sock ] ; then echo "oops" ; fi

if [ spreadsheet.csv -nt backup.csv ] ; then cp -f spreadsheet.csv backup.csv ; fi
```

Conditionals Using Regular Expressions

- Uses [[ ]] (instead of [ ]) with the =~ binary operator
- Quotes allowed to make a string literal

```
if [[ $variable =~ pattern ]] ; then …
if [[ foo =~ ^ba?r[[:space:]]* ]] ; then …

pattern='reg(ular)?[[:space::]*exp?(ression)?s?'
if [[ $string =~ $pattern ] ; then …
```

Variable Expansion: dealing with null / undefined variables

- ${variable:-default} : if variable is null or undefined, return default
- ${variable:=default} : if variable is null or undefined, assign default to variable
- ${variable:?message} : if variable is null or undefined, write message to STDERR and exit

```
unset status
printf "This presentation is %s\n" "${status:-awesome}"  # $status remains null
printf "This presentation is %s\n" "${status:=awesome}"  # $status becomes awesome

the_grade="${status:?Error: no status passed}"  # writes to STDERR and quits
```

Variable Expansion: substrings by offset

${#variable}	: returns the length of the variable
${variable:offset}	: returns from the offset to the end of the string
${variable:offset:length}	: returns from the office, but only length characters
negative offset	: count from the end of variable (needs a space after first colon)
negative length	: "length" becomes the offset from the end of the variable

```
string="0123456789"
printf "%s\n" "${#string}"          # 10
printf "%s\n" "${string:7}"         # 789
printf "%s\n" "${string:3:1}"       # 3
printf "%s\n" "${string: -4:2}"     # 67
printf "%s\n" "${string: 7:-2}"     # 7
```

Variable Expansion: substrings by pattern globs

${variable#pattern}	: from start, remove up to first occurrence of pattern
${variable:##pattern}	: from start, remove up to last occurrence of pattern
${variable:%pattern}	: from end, remove up to first occurrence of pattern
${variable:%%pattern}	: from end, remove up to the last occurrence of pattern

```
string="foobarbazzle"
printf "%s\n" "${string#*b}"          # arbazzle
printf "%s\n" "${string##*b}"         # azzle
printf "%s\n" "${string%b*}"          # foobar
printf "%s\n" "${string%%b}"          # foo
```

Substrings and filename manipulation

```
archive_filename="/path/to/file.ext"

filename="${archive_filename##*/}"        # strip to the last /
## file.ext

directory="${archive_filename%/*}"        # strip from the last / (note: removes last /)
## /path/to

filebase="${filename%%.*}"                # strip from the last . (note: removes last .)
## file

extension="${filename##*.}"               # strip up to the last . (note: removes last .)
## ext
```

Variable Expansion: regular expressions

${variable/pattern/string}	: replace first match of pattern with string
${variable//pattern/string}	: replace all matches of pattern with string
${variable/#pattern/string}	: pattern much be at the start of the string
${variable/%pattern/string}	: pattern must be at the end of string

```
string="foobar"

printf "%s\n" "${string/o/O}"        #  fOobar
printf "%s\n" "${string//oo/O}"      #  fOObar
printf "%s\n" "${string/#foo/FOO}"   #  FOObar
printf "%s\n" "${string/%bar/BAR}"   #  fooBAR
```

Declaring Functions

```
function_name() {
    function_body
}
```

Use local to set variable scope

```
list_executables() {
     local pattern='s/\*$//'
     ls -Fa | sed -Ene "${pattern}p"
}
```

Positional Arguments

- $n or ${n} where n is the number of the argument
- ${n} form is required when n >= 10 (e.g., ${10} is good, $10 is not)
- shift removes the first parameter in the argument list

```
second_word() {
     printf "%s\n" "${2}"
}

all_words() {
     while [ "$1" != "" ] ; do printf "you said '%s'\n" "$1" ; shift ; done
}
```

Environmental Arguments

- Create a variable outside of the function and call it from the inside

```
lower_case() {
     printf "%s\n" "${word:? No word passed}" | tr '[A-Z]' '[a-z]'

word="Foobar" lower_case
```

Named Local Arguments

- Use local "$@" to import arguments into a function's scope
- Allowing the caller to set functions' variables may have security implications

```
foobar() {
      local "$@"
      printf "%s\n" "${foo:-bar}"
}

foobar foo=bazzle
```

Returning Values with Exit Codes

- return ends the function; exit ends the program
- A exit code of 0 means success; non-zero indicates failure
- May be used to allow functions to serve as conditional operators

```
is_true() {
     if [ "$1" == "true" ] ; then return 0 ; else return 1 ; fi
}

if is_true "${variable}" ; then echo "Yay" ; else echo "boo" ; fi
```

Returning Flat Strings

- STDOUT from a function may be captured by the calling function

```
upper_case() {
     echo "$@" | tr '[:lower:]' '[:upper:]'
}

my_string="$(upper_case "$string")"
```

Returning Complex Values with jq

- Create a JSON object, send it via STDOUT, and have the caller parse it
- This isn't pure Bash – jq is an external dependency
- This may add a non-trivial amount of overhead
- This is more of a last resort – consider refactoring, caching, etc.

```
author() {
    jq -n '{name: {first: "Wes", last: "Dean"}}'
}

printf "My first name is '%s'\n" "$(author | jq -r '.name.first')"
```

Positional Command Line Arguments

- Works just like with functions (e.g., $1, ${2}, shift, etc.)
- Iterate across all arguments with for, in, and $@
- File globs, tilde expansions, etc. handled by the shell

```
if [ "$1" == "duck" ] ; then echo "Quack!" ; shift ; fi


for value in "$@" ; do
     printf "Received value '%s'\n" "$value"
fi
```

Using getopts with short flags

```
while getopts "n:w:" opt ; do
     case "$opt" in
          'n') number="$OPTARG" ;;
          'w') word="$OPTARG" ;;
          *  ) echo "Invalid option" 1>&2 ; exit ;;
     esac
done

shift "$((OPTIND - 1))
```

Using getopts with long flags

```
for arg in "$@" ; do
    shift
    case "$arg" in
        '--number') set -- "$@" "-n" ;;
        '--word')   set -- "$@" "-w" ;;
        *)          set -- "$@" "$arg" ;;
    esac
done

OPTIND=1
```

```
while getopts "n:w:" opt ; do
    case "$opt" in
        'n') number="$OPTARG" ;;
        'w') word="$OPTARG" ;;
        *  ) echo "Invalid option\n" 1>&2 ; exit 1 ;;
    esac
done

shift "$((OPTIND - 1))
```

Refactoring Code into Libraries

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

set -euo pipefail

library__some_function() {
     printf "yay"
}

main() {
     library__some_function
}

[[ "$0" == "${BASH_SOURCE[0]}" ]] && main "$@"
```

```
#!/usr/bin/env bash
set -euo pipefail

source "/path/to/some_library.bash"


for library in "lib/*.bash" ; do
  [ -e "${library}" ] && . "${library}"
done


library__some_function
```

Unit Testing with BATS

- BATS: https://github.com/bats-core/bats-core

```
setup() {
     load filename         ### NO EXTENSION
}

teardown() {
     rm -rf "${BATS_TMPDIR}"
}

@test "message to display" {
     run function_name    # thing to test
     [ "$status" -eq 0 ]  # expected result
}
```

```
@test "some other message" {
     run some_other_function 1 2 whatever
     [ "$output" = "whatever" ]
}

@test "another_function" {
     printf "# Some message\n" 1>&3
     run another_function some params
     [ "${lines[0]}" == "this should be line 1" ]
     [[ "${lines[1]}" =~ something ]]
}
```

Documenting with Doxygen

- bash-doxygen: https://github.com/Anvil/bash-doxygen

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

## @file filename
## @brief one-line description
## @details
## …
## @author author's name

set -euo pipefail

## @var varname
## @brief one-line description
```

```
## @fn function_name()
## @brief one-line description
## @details
## …
## @retval 0 condition
## @retval 1 condition
## @par Example
## @code
## …
## @endcode
```

Static Code Analysis with Shellcheck

- Shellcheck: https://github.com/koalaman/shellcheck
- Ignore issues with # shellcheck disable and/or SHELLCHECK_OPTS

```
# shellcheck disable=SC2059
printf "\x$1"

export SHELLCHECK_OPTS="-e SC2059"
```

Reformatting with shfmt

- shfmt: https://github.com/mvdan/sh
- Google' Shell Style Guide: https://google.github.io/styleguide/shellguide.html

```
shfmt -i 2 -ci -w /path/to/filename.ext
```

Safer Bash Programming Tips

- Use env to invoke Bash in scripts
  - #!/usr/bin/env bash
- Use set -euo pipefail near the start
  - -e : exit immediately if a command fails
  - -u : exit immediately if an undeclared variable is used
  - -o pipefail : return the right-most non-zero result code
- Declare all variables
  - Use local variables in functions when possible
- Check return values
- Know when to use " and '
- Use quotation marks
  - "${variables}"
  - "/path/to/filename.ext"
- Be wary of platform differences
  - External dependencies
  - Different paths / implementations
    - GNU grep vs BSD grep
  - Consider using built-ins (e.g., printf)
- Be kind to your future self
  - Write documentation
  - Write tests
  - Refactor common code into libraries

Shell Script Template

https://github.com/flexion/bash_shell_script_starter

Thank you!

...questions?

SQUIRREL!
NOW GET BACK TO WORK.
imgflip.com

This slide intentionally left blank.

Instructions for changing images

1. In the header, there's a spot to insert the name and picture of the Flexioneer who is presenting this slide
   a. This is NOT a part of the master slide - it will **need to be copy/pasted into each slide individually** so that it can be changed per slide
   b. Typically we use people's LinkedIn profile pics - any pics are ok, but make sure they are work-appropriate
   c. **To change the image**
      i. Right click on the "MySpace Tom" image next to Speaker name
      ii. Choose "Replace image" from dropdown
      iii. Select the image from your desktop, etc that should go in its place - you may want to crop in on the image to make sure it's mostly the person's face that's appearing
2. Use the same process for replacing images in the "team" slides

Bash, the Bourne Again SHell

- Bash is a shell – it's the thing you type into at the command line
- Maintained by the GNU Project; licensed under GPL 3.0
- Superset of Bourne Shell (sh)
- POSIX-compliant
- Still fairly new being only 33 years old (1989)

Comments

- Begin with words starting with '#'

```
# this is a comment

printf "foobar" # this is also a comment

printf "but # this is NOT a comment"

## @fn foobar()
```

Line Continuation

- Continue lines with backslash (\ ) as the **LAST** character of the line (no trailing spaces)

```
printf "This is a " \
"long line."

printf "This is a \
long line."

printf "This is a long line."
```

Quotes

- May be single, double, or "ANSI C" quoted; double quotes use bashslash (\ ) to escape

```
printf "The value of foo is $foo"       The value of foo is bar

printf 'The value of foo is $foo'       The value of foo is $foo

printf $'What will this print\?'        What will this print?

printf "The value of \$foo is '$foo'"   The value of $foo is 'bar'
```

Redirection

- Substitute output or input streams

```
printf "Hello world" > test.txt       # Overwrite test.txt with Hello world

printf "Hello world" >> test.txt      # Append Hello world to test.txt

printf "Hello world" &> /dev/null     # Send STDOUT and STDERR to /dev/null


sort < test.txt > sorted.txt          # Read test.txt, sort it, dump it to sorted.txt
```

Pipes

- | : Use STDOUT from one process as STDIN to another
- |& : same as | but also send STERR (same as 2>&1 |)
- mkfifo to create named pipes

```
cat test.txt | sort                      # send the contents of test.txt to sort
sort < test.txt                          # same
cat test.txt | sort | uniq > uniques.txt # same, but send unique lines to uniques.txt

mkfifo ~/mypipe                          # create a named pipe at ~/mypipe
  ls > ~/mypipe                          # put stuff into the pipe (blocks until read)
  sort < ~/mypipe                        # get stuff out of the pipe
```

File Handles

- 0 : standard input (STDIN)
- 1 : standard output (STDOUT)
- 2 : standard error (STDERR)
- *n* : arbitrary file handle

```
printf "foo" 1> test.txt           # send STDOUT to test.txt same as printf "foo" > test.txt

printf "foo" > /dev/null 2&>1      # send STDOUT and STDERR to /dev/null

printf "foo" 1> one.txt 2> two.txt # send "foo" to one.txt and any errors to two.txt
```

Stuff I wanted to talk about…

Bourne Again SHell

Comments

Line Continuation

Separating Commands

Declaring Variables

Naming Variables

Built-in Variables

Indexed and Associative Arrays

Iterating Across Indexed and Associative Arrays

Quotes

Redirection (>, >>, >&, <)

Filehandles (1: STDOUT, 2: STDERR, 0: STDIN)

Pipes (inline, named)

Conditions

Regular Expressions

Loops

Grouping commands (groups vs subshells)

Brace expansion, extglobs

Variable expansion

Tilde expansion

Math

Functions

Positional arguments

Getopts

Tests

Tools