Class 4

Shells

feat. Bash

```
:(){ :|:& };:
```

Do **NOT** run this

Overview

1. Announcements
2. Review + Exercises
3. Q&A
4. Basic assignment

Announcements

- Unix assignments: October 23
- Shell assignments: October 30
- Unix survey closing today
- Reference slides for each week!

Review

Review

- Command grouping
  - `(commands)`
  - `{ commands; }`

Expansion

Parameter expansion ("variable" expansion)

- `$varname`
- `${varname}`
- `${varname:-[value]}`: use default value
- Bash substring expansions
  - `${varname:offset}`
  - `${varname:offset:length}`

Expansion

Filename expansion ("glob"/"wildcards")

- Expand out to filepaths that match the pattern
- `*`, `?`, and `[]`

Command substitution (via subshell)

- `$(command)`: substitute the output of a *command* in the brackets
- Q: `$logname` vs. `$(logname)`?

Expansion

Arithmetic expansion

- `$((expr))` will expand to an evaluated arithmetic expression *expr*
  - Integer only

Process substitution (Bash)

- `<(command)` will substitute the *command* output as a filepath, with the output of *command* being **readable**
- `>(command)` will substitute the *command* input as a filepath, with the input of *command* being **writeable**
- `$ diff <(echo hello) <(echo olleh | rev)`
  - `diff` takes in two file names, but we're replacing them with "anonymous" files containing the command outputs

Exercises

1. Assign a variable `greeting` to a string that is concatenation of the string "user:" and the `USER` variable
2. Write a `mv` command that moves all files in the current directory that end in `.txt` into a directory called `text`
3. Use a command substitution (`$(commands here)`) to get the output of `whoami` and save it into a variable `me`

Quoting

- Single quotes (`'`) preserves **all** of the characters between them
- Double quotes (`"`) preserve all characters except: `$`, `\`, and backtick

Compound commands and control flow

`if-elif-else`

```
# '#' comments out the rest of the line
# elif and else are optional parts
if test-commands; then
  commands
elif more-test-commands; then
  more-commands
else
  alt-commands
fi
```

- *test-commands* is executed and its **exit status** is used as the condition
  - *0* = success = "true", everything else is "false"

Commands for conditionals

- `test expr`: `test` command
- `[ expr ]` (remember your spaces! `[` is technically a utility name)
- `test $a -eq $b`
- `[ $a -eq $b ]`
- These set the exit status (`?`) to 0 (true) or 1 (false)
- `[ $a -eq $b ] && [ $a -lt 100 ]`
- `test $a -eq $b && test $a -lt 100`

Commands for conditionals (Bash)

- `[[ expr ]]`: **Bash** conditional
  - Richer set of operators: `==`, `!=`, `<`, `>`, among others
  - **Note**: The symbol operators above operate on strings
- `((expr))`: **Bash** arithmetic conditional
  - Evaluates as an arithmetic expression
  - `(($a < $b))`: this would evaluate to "false" if a=100, b=2

`while`

```
while test-commands; do
  commands
done
```

- Similarly to `if`, the exit status of *test-commands* is used as the conditional
- Repeats *commands* until the condition **fails**

`until`

```
until test-commands; do
  commands
done
```

- Repeats *commands* until the condition **succeeds**

`for`

```
for var in list; do
  commands
done
```

- Each iteration *var* will be set to each member of the *list*
- *list* is simply a list of whitespace-delimited strings
- *list* will have any necessary expansions performed
- Note: if there is no `in list`, it will implicitly iterate over the argument list (i.e. `$@`)
- Example lists:
  - `1 2 3 4 5`
  - `$(ls)`
  - `$(seq 1 5)`

`case`

```
case value in
  pattern1 ) commands1 ;;
  pattern2 ) commands2 ;;
  multpat1 | multpat2 ) commands3 ;;
  * ) commands
esac
```

- *value* is matched against patterns
- When a pattern is matched its command(-list) is run
- A wildcard pattern is often used to represent a "default" case

Exercises

1. Write an `if` statement that prints "success!" if the last command ran successfully
   - Remember the `?` variable?
   - `echo` can print text for you
   - `true` and `false` can give you a success and failure
2. Write a `for` loop that creates 5 files, named `file1` to `file5`
   - `seq 1 5` can produce a list of integers from 1 to 5
   - `touch` can create empty files for you

Functions

```
func-name () compound-command # parens are mandatory
# or
function func-name () compound-command # [Bash]; parens are optional
```

- A **compound command** is a **command group** (`()`, `{}`) or a control flow element (`if-elif-else`, `for`)
- Called by invoking them like any other utility, including **passing arguments**
  - Arguments can be accessed via `$n`, where *n* is the argument number
  - `$@`: list of arguments
  - `$#`: number of arguments

Examples

```
hello-world ()
{
  if echo "Hello world!"; then
    echo "This should print"
  fi
}
# calling
hello-world
```

```
# Bash
function touch-dir for x in $(ls); do touch $x; done
# calling
touch-dir
```

```
echo-args ()
{
  for x in $@; do
    echo $x
  done
}
# calling
echo-args a b c d e f g
```

```
# Bash
function divide
{
  if (( $2 == 0 )); then
    echo "Error: divide by zero" 1>&2
    # the redirection copies stderr to stdout so when echo
    # outputs it's really going to the caller's stderr
  else
    echo $(($1 / $2))
  fi
}
# calling
divide 10 2
divide 10 0
```

Configuring the shell

- Shells will automatically source certain files to perform configuration
  - `/etc/profile`: system-wide configuration
  - `~/.bashrc`: Bash's user shell configuration file
  - `~/.zshrc`: Zsh's user shell configuration file
- You can make your own additions to your `~/.bashrc` or `~/.zshrc` etc.
- Maybe you want to add a directory to `PATH`?
  - `export PATH="newdir:$PATH"`
- Maybe I want to alias a word to a command that navigates to my Windows side?
  - `alias cdw='cd /mnt/c/Users/brandon/'`
- Maybe I want to change up my prompt?...
  - `PS1` variable

Q&A

Basic assignment