Why Shell Scripting

- Bash shell scripts
- Execute bash commands from a file
- Automate sequences of shell commands
- Run them later or at scheduled times
- Take advantage of UNIX toolset

Why Shell Scripting

- Who needs it?
  - System administrators
  - Developers
  - Power users

Why Shell Scripting

- Bash scripts are very good at:
  - File manipulation
  - Running programs
  - Processing text

Why Shell Scripting

- Sometimes other languages are better
  - Examples: mathematical calculations, binary data, graphics

What we will cover

- All the basics of shell scripting
  - I/O
  - Variables
  - Conditional execution (if, then, else, case)
  - Loops
  - Arithmetic

What we will cover

- All the basics of shell scripting
  - String manipulation
  - Handling script arguments
  - Shell functions

Creating a Script

- A script:
  - A text file containing code
  - To be run by an interpreter
  - In this course, the interpreter will be bash
  - Will run each command in the file in order

Use a good text editor

- Emacs, vi
- Linux: Kate, gedit
- Mac OS: TextWrangler

File permissions

- Use chmod command:
  - “chmod u+x filename”
  - “chmod a+x filename” to make it executable for everyone
  - “chmod a-x filename” to remove the permission

Calling the script

- If the script is not on your PATH:
  - Include the directory when calling it
  - ./script1.sh
  - /home/bbrelin/script1.sh

Calling the script

- If the script is on your path
  - Just call it like a regular command
- Tip: make a bin folder in your home
  - Put your scripts in there
  - Add it to your path
  - PATH=”$PATH:~/bin”

The Shebang operator

- Shebang line:
  - First line of file
  - Starts with #!
  - Specifies which interpreter should run the code
  - Specify options for interpreter

The Shebang operator

- Bash scripts:
  - #!/bin/bash
- Other systems than Linux or Mac OS
  - May have bash in a different location
  - #!/usr/bin/env bash
  - This will find bash on the user's PATH
  - Cannot give options
  - Result depends on the user's configuration

Naming your shell script

- Don’t name your script “test”, “if”, or “ls”
- Conflicts with existing commands
- Does a command with the same name exist?

Naming your shell script

- Don’t name your script “test”, “if”, or “ls”
- Conflicts with existing commands
- Does a command with the same name exist?

Shell Variables

- Used to store data by name
- To create: just assign a value
- x=10
- If x already existed, it is assigned the new value
- filenames=”example1.txt example2.pdf example3.jpg”
- Values containing spaces: use quotes
- Don’t use whitespace around =

Shell Variables

- To get the value
- Prefix with $
- For example: echo $x
- Bash variables have no type
- Basically just store a string

Variable Names

- Names:
  - Only letters, numbers, and underscore are allowed
  - First character should be a letter or an underscore
  - Variable names are case-sensitive
- Uppercase variables:
  - Bash has many pre-defined variables
  - PATH, HOME, SECONDS, IFS, etc.
  - You don't want to override them by mistake
- Good Habit:
  - Use lowercase names for your variables

Using Shell Variables

- Good habit: surround your variables with quotes
  - Use “$x” instead of $x
  - Prevent surprises when it contains spaces
  - Use double quotes: keep meaning of dollar sign intact
- Braces
  - Where does your variable name end?
  - echo “${foo}bar”
  - prints value of var “foo” followed by string “bar
  - echo “$foobar” prints value of “foobar”
- Using braces a lot is a Good Habit
- Another good habit
  - Use $HOME instead of ~

Reading Input

- Reading Input
- read
  - Reads a line of input into a variable
  - read var
  - Is a shell builtin
  - “help read”
  - “man builtins”
  - read -p “Type your name: “ name

Control Flow with ‘if’

Code examples

```
if some_expression; then
    some_code_executed_here
fi
```

Control Flow with ‘if’

Code examples

```
if some_expression; then
    some_code_executed_here
elif some_other_expression; then
    some_other_code_executed_here
else
    some_other_code_executed_here
fi
```

Control Flow with ‘if’

- Keywords if, then, else, fi
- First on a line, or
- After a semicolon
- help if

Test operations

- Conditional Expression
  - Tests on files and directories
  - Tests on strings
  - Arithmetic tests

Test operations

Expression	True if
[[$str]]	str is not empty
[[$str = “something”]]	str equals string “something”
[[$str=”something”]]	always returns true!
[[-e $filename]]	file $filename exists
[[-d $dirname]]	$dirname is a directory

Test operations

- Spaces around the expression are very important!
- Same for switches (-e) and equals sign

More conditional operators

- Classical command: “test”
- Also: [
- Harder to use, easier to make mistakes
- Only use for portability
- [[…]] is a bash extension
- Not a command but special syntax
- No quotes needed around variables
- Good habit: use [[..]] instead of [..]
- Getting help
- “help test” will show you most important info
- “help [[“ will tell you about the extension

Arithmetic tests

- Arithmetic tests
- For comparing integers only
- [[arg1 OP arg2]]
- Where OP is:
- o -eq: equality
- o -ne: not equal
- -lt: less than
- -gt: greater than
- And some others.. see help
- So don’t use =, >, < for numbers!

Special test variables

- Special variables:
- $# contains number of script arguments
- $? contains exit status for last command
- To get the length of the string in a variable:
- Use ${#var}

More code examples

```
if [[ ! -d $bindir ]]; then
    #If not: create bin directory
   if mkdir $bindir; then
       echo 'Created bindir'
   else
       echo 'Could not create bindir'
   fi
fi
```

More code examples

```
if [[ $count_1 -gt $count_2 ]]; then
   echo '${dir} has most files'
else
    echo '$dir2] has most files'
fi
```

And, or and not in if statements

- In a conditional expression:
  - Use ! to negate a test:
    - [[! -e $file]]
    - Use spaces around !

- Use && for “and”:
- [[$#-eq 1 && $1 = “foo”]]
- True if there is exactly 1 argument with value “foo”

- Use || for “or”:
  - [[$a || $b]]
  - True if a or b contains a value (or both)

The Echo command

Echo

- Prints its arguments to standard output, followed by a newline
- -n suppresses the newline
- -e allows use of escape sequences
  - \t: tab
  - \b backspace
- These options are not portable to non-bash shells

The printf command

- printf
  - Can do more sophisticated output than echo
  - Uses a format string for formatting
  - Will not append a newline by default

- For help:
  - help printf
  - man printf
  - man 3 printf
  - -v will send output to a variable

More on the read command

- read can read words in a line into multiple variables
- read x y
- input “1 2 3”: x=1, y=”2 3”
- Uses IFS variable for delimiters

More on the read command

- Reads input into a variable
- “read x”
- No variable specified? Will use REPLY
- -n or -N specifies number of characters to read
- -s will suppress output (useful for passwords)
- -r disallows escape sequences, line continuation
- Good Habit: always use -r
- Several more useful options (see help)

UNIX I/O streams

- Three standard streams
- input, output, error
  - Represented by number (file descriptor), or special file
  - 0: Standard Input (stdin)
- /dev/stdin
  - 1: Standard Output (stdout)
- /dev/stdout
  - 2: Standard Error (stderr)
- /dev/stderr
  - Used for diagnostic or error message
- /dev/null discards all data sent to it

Redirection of I/O

- Get input from somewhere else, send output or errors somewhere else
- Input redirection: <
- grep milk < shoppingnotes.txt
- Output redirection: >
- ls > listing.txt
- Will overwrite existing files!
- (although this can be customized with the set command)
- >> appends to the end of a file
- Pipes
- ls | grep x

Redirection of I/O

- Redirect a specific stream with N>
- “cmd 2> /dev/null” discards all errors
- Redirect to a specific stream with >&N
- >&2 sends output to stderr (equivalent to 1>&2)
- 2>&1 redirects stderr into stdout
- Sending both error and output to a single file
- cmd > logfile 2>&1
- Don’t do this: cmd > logfile 2> logfile
- Allowed anywhere on the command line, but order matters
- cmd < inputfile > outputfile
- >outputfile cmd < inputfile
- cmd >logfile 2>&1 (sends errors to the logfile)
- 2>&1 >logfile cmd (sends errors to stdout)

Flow control with loops

- Loops
- while/until
- for
- break and continue
- case
- Compound commands
- || and &&

Flow control with loops

```
while some_expr_is_true; do
    do_some_code_here
done
```

- Repeats code in the block
- Continues as long as test returns true

Flow control with loops

```
until some_expr_is_true; do
    do_some_code_here
done
```

- Repeats code in the block
- Continues while the test returns false

Flow control with loops

```
for some_var in some_values; do
    do_some_code_here
Done
```

- Assign each value in some_values to some_var in turn
- Will stop when no words are left
- Do NOT quote SOME_VALUES

Flow control with loops

```
for (( INIT; TEST; UPDATE)); do
    do_some_code_here
done
```

- C-style for loop
- Use double parentheses
- First expression: initialize your loop variable(s)
- Second expression: a test. The loop will run as long as this is true
- Third expression: update the loop variable(s)

The break and continue statements

- break
  - quits the loop
- continue
  - skips the rest of the current iteration
  - continues with the next iteration
- Both can be used in for, while and until

Flow control with case

```
case some_value in
    PATTERN1)
        do_some_code_here
        ;;
    PATTERN2)
        do_some_code_here
        ;;
    PATTERNn)
        do_some_code_here
        ;;
    *)
        do_default_action_here
        ;;
```

Flow control with case

- Matches word with patterns
- Pattern matching is the same as with matching filename patterns
- Use *, ?, []
- Code for first pattern that matches gets executed
- End code with ;;
- so you can use multiple statements separated by ;
- Multiple patterns separated by |

Grouping commands

- Group commands with {}
- Will group them into a single statement
- Can use I/O redirection for the whole group
- Use the group in an if statement or while loop
- Return status is that of the last command in the group
- { cmd1; cmd2; cmd3; }
- Separate the commands with newlines or semicolons
- Use spaces around braces
- Ending semicolon or newline not optional!

Using && and ||

- Execute next statement depending on return status of previous
- statement
- Basically: short for if
- &&
- Will execute next statement only if previous one succeeded
- mkdir newdir && cd newdir
- ||
- Will execute next statement only if previous one failed
- [[$1]] || echo “missing argument” >&2
- [[$1]] || echo “missing argument” >&2 && exit 1
- Dont do this: will always exit!
- [[$1]] || { echo “missing argument” >&2; exit 1; }

Integer operations

- Integer variables
- declare -i num
- Now $num can only hold numbers
- Trying to set it to something else will NOT give an error
- Instead, this will set a value of 0
- Unset an attribute with +
- declare +i num
- Triggers arithmetic evaluation

Integer operations

- C-like syntax for doing calculations
- § let command
- o
- let n=100/2
- ((..))
- ((++x))
- ((p=x / 100))
- ((p= $(ls | wc -l) * 10))
- This is a command equivalent to let
- $((..))
- This a substitution, not a command
- p=$((x / 100))
- With a variable declared as an integer
- num=”30 % 8”

Integer operations

- No need to quote variables
- ((..)) can be used in if, while
- 0 is false, anything greater than 0 is true
- ((0)) || echo “false”
- Pitfall: numbers with leading zeros are interpreted as octal
- So 010 = 8
- ((..; ..; ..)) syntax in for loop is NOT an arithmetic expression
- but the three expressions separated by ; are

Exporting variables

- By default, variables are local to your script or terminal session
- Export a variable
- To make it available to subprocesses
- You cannot pass a variable to the program that runs your script
- export var
- export var=”value”
- declare -x var
- declare -x var=”value”

Shell special variables

- Positional Parameters
- Hold the n-th command line argument: $1, $2, etc.
- Above 9 use braces: ${10}, ${25}
- $0
  - Holds name of the script as it was called

Shell special variables

- $@: All
- Equivalent to $1 $2 $3 ... $N
- But when double quoted: “$1” “$2” “$3” ... “$N”
- So parameters containing multiple words stay intact
- $*
- Equivalent to $1 $2 $3 ... $N
- But when double quoted: “$1 $2 $3 ... $N”
- Don’t use this; use $@ instead!
- $#
- Holds the number of arguments passed to the script

The shift statement

- Removes the first argument
- All positional parameters shift
- $2 -> $1
- $3 -> $2
- $4 -> $3
- etc.
- $# lowered by 1
- Give a number to shift multiple:
- shift 3 removes the first three arguments

Shell functions

- Define your own command
- name () { ... }
- You can run the code in the braces as a new command
- other equivalent syntax (not recommended):
- function name () { ... }
- function name { ... }
- Execute it like any command
- Give it arguments
- Use redirection
- Positional parameters are available for function arguments
- $1, $2, …
- Naming your functions
- same rules as for naming scripts: don’t override existing commands

Shell functions

- Bash variables are globally visible
- In a function, you can make a variable local to that function
- Use declare or local
- Exit a function with return
- returns a status code, like exit
- Without a return statement, function returns status of last statement
- Returning any other value
- Use a global variable
- Or send the data to output and use command substitution
- Exporting a function
- export -f fun