Bash and Bash Scripting

Start of Week 3

Bash is a shell and command language used as a replacement for the Bourne Shell – called the Bourne-again Shell. Widely available – there may be some linux machines that don’t have bash pre-installed.

* <http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html>
* <http://www.tldp.org/LDP/Bash-Beginners-Guide/html/>
* <https://ostechnix.com/list-useful-bash-keyboard-shortcuts>

Most of what you can do in a bash script you can also do from the command line. A lot of “bash one-liners” are essentially small scripts.

## Navigation, repeating commands, etc: (Can’t use in scripts)

* Arrow keys can be used to go up and down through previous commands
* Ctrl-R to search our command history, Ctrl-P, Ctrl-N to navigate
* Alt + . use the last word of the previous command
* !! for the last command.
  + Text

    Description automatically generated
* !# in combination with the history command to execute a certain historical command
* !xyz repeasts the last command starting with xyz
* !$ give you the last word of the previous command

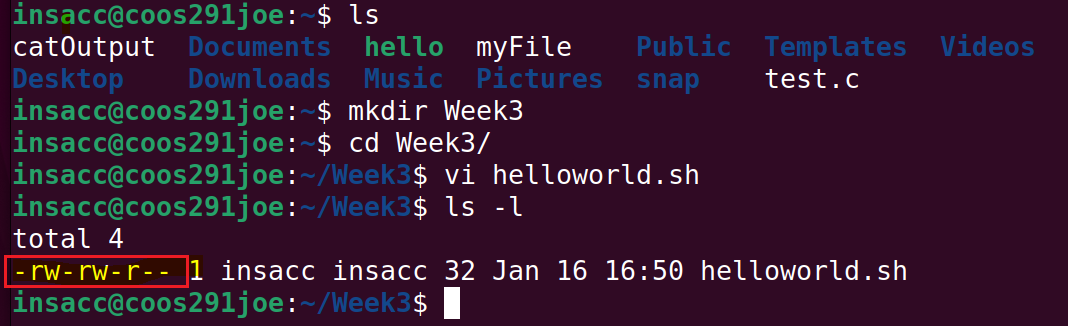
## Your First Bash Script (Hello World)

* Need the shebang (#!/bin/bash)
* File extensions of our bash script do not matter – some people will use .sh or .bash or something else to distinguish bash scripts from other files.
* We often use commands whereas in say, Java, we’d use methods.
* We have to change the permissions on the bash script in order to execute it. We would use the **chmod** command to change this:
  + <https://www.linuxfoundation.org/blog/blog/classic-sysadmin-understanding-linux-file-permissions>
* We also have to list the location of the interpreter (/bin/bash)

Our first script:

|  |
| --- |
| #!/bin/bash  echo “Hello World!” |

## Permissions

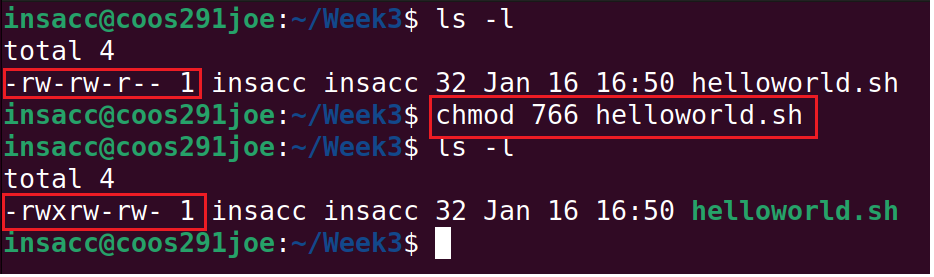


Taking a look at **helloworld.sh** via the long listing (ls -l), we can see we have 10 characters on the left that signify the type of file (- represents a file, d a directory, etc), and the remaining 9 characters correspond to the owner, group, and all users permissions.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Owner** | | | **Group** | | | **All Users** | | |
| - | R (4) | W (2) | * (1) | R | W | - | R | - | - |
| This is a file | Read | Write | No permission to execute | Read | Write | No permission to execute | Read | No permission to write | No permission to execute |

Using the **chmod** command, we can alter the permissions for a file/directory in terms of what the owner, group, and all users have access to do. If we want to execute our helloworld.sh, we would need to add the “execute” permission to that file, via the command **chmod 766 helloworld.sh**

|  |  |  |
| --- | --- | --- |
| **7** | **6** | **6** |
| Full permissions for owner (4+2+1) | Read/Write permissions for group  (4+2) | Read/Write permissions for all users  (4+2) |

****

**To execute:**

**Text

Description automatically generated**

## Scripts you’re already running

~/.bash\_profile (or .profile) and ~/.bashrc are shell scripts that contain shell commands. These files are executed in a user’s context when a new shell opens or when a user logs in so that their environment is set correctly.

~/.bash\_profile (or .profile) is executed for login shells and ~/.bashrc is executed for interactive non-login shells. This means that when a user logs in (via username and password) to the console (either locally or remotely via something called SSH), the ~/.bash\_profile (or .profile) script is executed for the initial terminal and returned to the user. After that, every time a new shell is opened, the ~/.bashrc script is executed. This allows users more fine-grained control over when they want certain commands executed. These shell scripts are meant to be written to by the local user to configure their own environment.

## Bash “Tricks” that work in scripts

At it’s base, a script is just a list of commands. Here are some things you can do from both scripts and command line:

* Brace expansion – “touch file{a,b,c}”, “touch file{1,2,3}”
  + touch filea
  + touch fileb
  + touch filec
  + Text

    Description automatically generated
  + Quickly copy a file to a backup
    - cp filename{,.old}
      * cp filename filename.old
      * Text

        Description automatically generated
  + & is used to fork a command (run in the background)
    - Running “firefox” from within our terminal will execute Mozilla Firefox as a process in the current session. Notice that we can’t interact with the terminal until Firefox is closed.
    - Running “firefox &” from within our terminal will run the same thing, but as a background process. We can still interact with the terminal.
  + ; is a command separator – “mkdir test; touch test/foo”
    - This will first create a directory called “test”
    - Then it will change the timestamp of the file “foo” within that “test” directory.
    - If that file doesn’t exist, touch creates it.
    - Graphical user interface

      Description automatically generated
  + && to execute ONLY IF the first command succeeds, || only if the first command fails
    - “cd foo && touch bar”
      * Touch bar (second command) will only execute if cd foo (first command) succeeds
    - “cd foo || mkdir foo && cd foo && touch bar”
      * If cd foo fails, it will execute mkdir foo
      * If mkdir foo succeeds, execute cd foo
      * If cd foo succeeds, execute touch bar
    - Be wary of using long chains like above, use if statements for that usually.

## Flow Control Basics

* Bash has an if statement!
* If \*test\*; then \*result\*; else \*other result\*; fi
* Text

  Description automatically generated
* There’s a handy-dandy “test” command to make logical decisions, as well as [ and [[ which we’ll talk more about later

## Variables

* $ means variable substitution
* Some environment variables, like $PATH, some special variables like $? And $$ will be covered later.
* We can set our own variables – name=Fred (Make sure we have no spaces)
* Calendar

  Description automatically generated with medium confidence
* New versions of bash support one-dimensional arrays. Declare it like “my\_array[2]=5” or “my\_array=(a b c d)”
  + But how do we reference them?
  + Text

    Description automatically generated
  + Have to use curly braces - ${my\_array[2]}
  + Can also @ to get the whole array, or #array[@] for length
* Use of ${} generally considered good practice
  + name=Fred
    - echo $nameerick
    - echo ${name}erick
    - Text

      Description automatically generated
* Bash variables are untyped. There is a “declare” but it’s very weak and we don’t typically use it at this stage.

## Arithmetic

* Variables can contain numbers, and we can do simple math.
* We use $(())
* Only use integers, though there is a calculate app (program) called “bc”
* Variables don’t need to be de-referenced in arithmetic, but you can if you want.

## Quoting, Escaping, etc (and relation to filename expansion)

* Dealing with spaces – how do you change to the director “foo bar”
  + foo\ bar
  + Don’t put spaces in directory names, generally.
  + Text

    Description automatically generated with medium confidence
* Difference between double (partial) and single (strong) quoting: Double quotes preserve meanings of things like $, `, and \, whereas single quotes only recognize ‘
* Quoting and escaping a very, very frequent sources of errors in scripts. Filename expansion will creep in, or an argument with unexpected spaces – so it is a good idea to check for problems in quotes first.
* With some utilities, like “echo”, \ will trigger a special meaning for a character – for instance \n for newline.

## Sample Function to demonstrate scope:

Text

Description automatically generated

## Looping Structure Examples

|  |
| --- |
| #!/bin/bash  for i in $( ls ); do  echo item: $i  done |

Text

Description automatically generated

Text

Description automatically generated

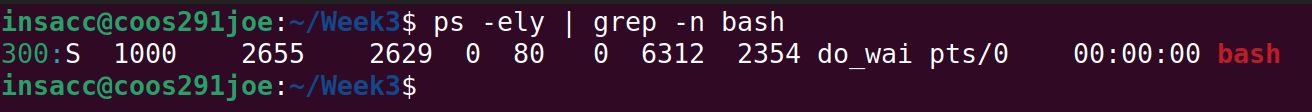
Text

Description automatically generated

Text

Description automatically generated

## Revisiting Pipes

* “Unix philosophy urges the user of small, yet highly-focused programs that can be used together to perform complex tasks.” – Part of the bash paradigm
* Similar to redirects (>, >>, <), but instead of redirecting files, they redirect stdout of one process to the stdin of another.
* Example: ps -ely and grep <process I’m interested in>
* Solution: ps -ely | grep -n bash
* 

## Process Substitution

“ps -ely | grep bash” will do more or less the same thing as “grep bash < (ps -ely)”

* A few commands take inputs from multiple places, but many will take multiple files.
  + diff <(ls -1 -s <dir1 name>) <(ls -1 -s <dir2 name>)
  + Text

    Description automatically generated

## Order of Operations

* There is an overall order expansions are done in – brace expansions, tilde expansions, (variable process substitutions, arithmetic expansion, command substitution done in left-to-right fashion), word splitting, and filename expansion.
* <http://mywiki.wooledge.org/BashParser>

## Alias

* The “alias” command allows us to define our own commands, with desired switches.
* Text

  Description automatically generated
* Try the following:
  + alias hellothere=”echo Hello World”
  + Text

    Description automatically generated
* Try the following
  + alias mymod=’chmod 766 `ls -t \*.sh | head -1`’
  + Here, I am grabbing the first line (head -1) from a reverse listing of .sh files (according to time updated), and then feeding that line (filename) to chmod

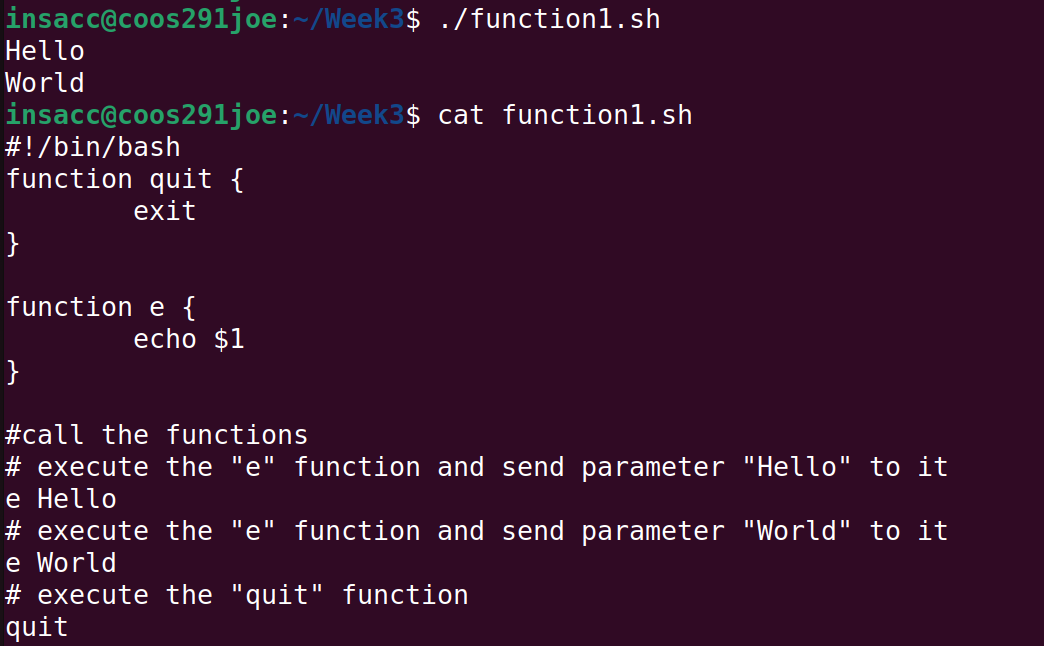
Text

Description automatically generated

## Functions

* We define functions in bash scripts via the keyword “function”
* Surround the scope for the function with braces – { }
* Call the functions by specifying the function name

|  |
| --- |
| function quit {  exit  }  quit |



* Parameters can be accessed via the “$” character
* For example, to refer to the first parameter, we would use “$1”, the second “$2”, etc

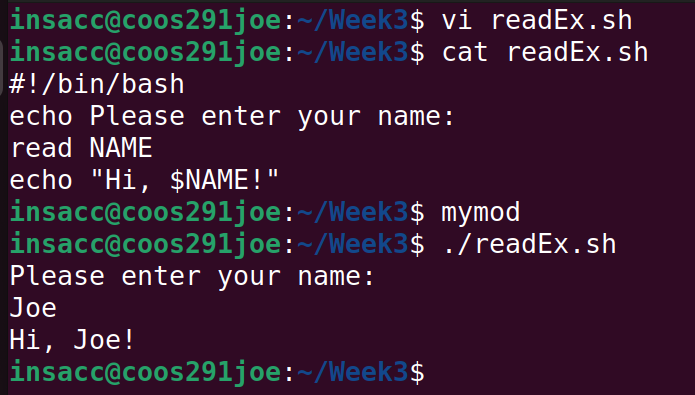
Menu Example

Text

Description automatically generated

## Read User Input with “read”

On many occasions, you may want to prompt the user for some input, there are several ways to achieve this. We can use the keyword “read” to read from STDIN, and also use it in combination with an identifier to save the input for later use.



Multiple inputs:

Text

Description automatically generated

Comments

* The line “read FN LN” has a whitespace character between variables FN and LN
* This can be matched with our whitespace character in our input. IE, “Joe Herbert” will detect the whitespace character between the two names and match accordingly.
* REMEMBER: If you want access variables, use the $ symbol followed by the variable name.

Let’s work together to create a new script that gives us a recursive directory / file listing and provide timing information on how long that listing took.

Remember, building up big, long command lines is useful – BUT sometimes we want to execute it from a file for re-use. At it’s simplest, bash scripts are just putting command in a file.

Let’s create a file called **ls\_time.sh** and open it in your editor of choice.

Text

Description automatically generated with low confidence

* We need to define variables before we use them
* ; are used to separate multiple commands
* > is used to redirect output to a file or other destination
* 2> redirects any errors that are encountered and puts them in a file specified by error\_log
* && and || are used for conditional execution of commands based on whether the command before was successful or not.

## Convert this script to use an IF block

* It would be handy if we could specify the directory we were going to do our **ls -R** from on the command line. What’s our of passing information to a program from the command line?
* We can use parameters when executing our script, and access them within the script via:
  + $1
  + $2
  + $3, etc
  + $# - # is the number of the parameter

Text

Description automatically generated

Comments:

* target\_directory=$1
  + This creates a new variables called “target\_directory” and assigns it the value of our first parameter that we used when executing the script. In this case, it was “~”.
* The if statement tests whether these commands (single command really, with redirected output for stdout and stderr). If they are successful, we enter the if statement execution block.
  + If unsuccessful, we enter the “else” block.

Maybe we should check that our target is actually a directory. For this, we can take a look at the “test” command. Tests are performed using [ ]. We can copy our ls\_timer2.sh file to ls\_timer3.sh as a starting point.

|  |
| --- |
| #!/bin/bash  # Does a recursive directory listing and times it.  directory\_listing=my\_dirs  error\_log=my\_errs  target\_directory=$1  if [ ! -d $target\_directory ] # tests whether our directory is in fact a directory, ! negates  then  # not a directory  echo "$target\_directory is not a valid directory. Need to specify something valid"  exit 1  fi  echo "Doing a recusive listing of $target\_directory"  start\_time=$(date +%s%N)  if ls -R $target\_directory > ${directory\_listing} 2> ${error\_log}  then  echo "Done. Took $((($(date +%s%N) - start\_time)/1000000)) ms."  else  echo "Could not list all directories."  fi |

Text

Description automatically generated

Comments:

* [ ! -d $target\_directory ] invokes the test command on our variable
  + ! negates the result, so we’re really testing if target\_directory is **NOT** a directory
  + -d is a switch for the test command, specifying that we’re testing if it’s a directory
  + $target\_directory is the location we wish to test.
  + exit 1 is a command to exit our script without errors.

We should probably make it so that we can have arguments for directory\_listing and error\_log

Text

Description automatically generated

Text

Description automatically generated

Comments:

* [[ $# -gt 1 ]] tests the number of parameters we have
  + “shift” will decrease this count
  + $# is the number of parameters
  + -gt means “greater than”
* shift moves the pointer of the first parameter to the next parameter
* [[ $key == “-l” ]] tests whether the value of our parameter is “-l”
  + If it is, grab the value of the next parameter and assign it to “directory\_listing”
    - Recall, that at the beginning of the while loop, we shifted 1 position.
* [[ $key == “-e” ]] tests whether the value of our parameter is “-e”

## Other Types of Loops

Let’s look at some different types of loops. We have break and continue just like in Java, as well as a select statement.

Text

Description automatically generated

Comments:

* trap is similar to catch in Java. For any interrupt keys we press, we’ll execute the echo statement followed by exit 1
* for element in \*
  + iterates through all files in our \* listing
  + then performs the command “file” on each, which outputs file descriptions for each file encountered

Text

Description automatically generated

Comments:

* my\_array=(a b c d e)
  + declares an array variable and assigns elements a,b,c,d,e to it
* for element in ${my\_array[@]}
  + iterates through each element of the array, regardless of size of the array
* echo “For loops: $element”
  + outputs the element value

Text

Description automatically generated

## January 23rd – Student work

Please go through the following steps for the first hour of class.

Comments (from loop3.sh):

* $#{my\_array[@]}
  + This returns the size of the array (number of elements)
* for ((i=0; i < ${#my\_array[@]}; i++))
  + This will create a for loop, with our iterating index as “i”
  + We initialize i to be equal to 0
  + We return the size of the array as our upper bound of our for loop (middle parameter)
  + At the end of each iteration, we increase i by 1 (i++)
* ${my\_array[$i]}
  + Specifies the value of my\_array at index i

### loop4.sh

Create loop4.sh in VI (or an editor of your choice) with the following code:

|  |
| --- |
| #!/bin/bash  trap "echo Quitting gracefully ...; exit 1" SIGINT  my\_array=(a b c d e)  echo "Number of elements in our array: ${#my\_array[@]}"  while\_counter=0  while [[ $while\_counter -lt ${#my\_array[@]} ]]  do  echo "While loop: ${my\_array[$while\_counter]}"  let "while\_counter++"  done |

Comments:

* while\_counter=0
  + declares a variable to be used in our while loop and sets the initial value to 0
* while [[ $while\_counter -lt ${#my\_array[@]} ]]
  + Again, this is just an expression that evaluated to a Boolean value (true or false)
  + Read as: While our counter is less than the size of our array, perform the following (while scope)
  + $while\_counter is the value of our counter variable, first run will have a value of 0
  + -lt stands for less than, we wouldn’t use < > in this case since those are reserved for Linux input/output redirects (stdout, stdin)
  + ${#my\_array[@]}
    - Returns the size of the array, in this case, 5 elements total
  + ${my\_array[$while\_counter]}
    - Takes the integer value of while\_counter variable and uses that as the index of our my\_array array.
  + let “while\_counter++”
    - the let command evaluates an arithmetic expression
    - in this case, it executes / evaluates “while\_counter++”
      * This adds 1 to the value of while\_counter

### loop5.sh

Create loop5.sh in VI (or an editor of your choice) with the following code:

|  |
| --- |
| #!/bin/bash  trap "echo Quitting gracefully ...; exit 1" SIGINT  my\_array=(a b c d e)  echo "Number of elements in our array: ${#my\_array[@]}"  until\_counter=${#my\_array[@]}  until [[ $until\_counter -lt 0 ]]  do  echo "Until loop: ${my\_array[$until\_counter]}"  let "until\_counter--"  done  select filename in \*  do  echo "You picked $filename."  done |

Comments:

* until\_counter=${#my\_array[@]}
  + This declares a variable called until\_counter and assigns it the size of the array my\_array (in this case, integer 5)
* until [[ $until\_counter -lt 0 ]]
  + Specifies that we wish to loop within our until scope until the statement is evaluated as true
  + Loop the following code until our $until\_counter is less than 0
* let “until\_counter--"
  + “let” will evaluate the arithmetic expression “until\_coutner--"
* select filename in \*
  + this creates a “menu” of options for the result of “\*” (file listing in current directory)
  + allows you select an option via the corresponding integer value
  + typing “q” with throw an “exception”, which will be “trapped” by the trap line.
    - If this occurs, it will execute the following statements:
      * echo “Quitting gracefully….”
      * exit 1

### Case and Boolean Operators

Here is some example code, call it case1.sh

|  |
| --- |
| #!/bin/bash  # Demonstrate flow control in bash  # We're taking one arguement  key=$1  # Using boolean operators  test $1 = "Yes" && echo "You said yes" || echo "You said something other than yes." |

**Try executing the bash script via:**

* **./case1.sh Yes**
* **./case1.sh yes**
* **./case1.sh No**

Comments:

* key=$1
  + takes our first parameter (when we ran the script) and assigns the value to a new variable called “key”
* test $1 = “Yes”
  + equivalent to [[ $1 = “Yes” ]]
  + This tests the value of $1 and compares it to the RHS “Yes”
* && echo “You said yes”
  + This statement is executed only if the parameter that we entered was “Yes”
* || echo “You said something other than yes.”
  + This statement is executed only if the parameter that we entered was something other than “Yes”
* Remember, strings are case sensitive (“Yes” vs “yes”)

Another example (case2.sh)

|  |
| --- |
| #!/bin/bash  # Demonstrate flow control in bash  # We're taking one arguement  key=$1  # If statement  if test $1 = "Yes"  then  echo "You said yes"  elif test $1 = "No"  then  echo "You said no"  else  echo "You said something else ($1)"  fi |

Comments:

* if test $1 = “Yes”
  + evaluates to true or false, depending on the value of the parameter sent to the script
* elif test $1 = “No”
  + Equivalent to Else If in other languages
  + If the first “If” Boolean test is false, it will look for elif statements to evaluate next until a true is encountered
* else
  + if nothing has been tested to true, this will execute.

Another Example (case3.sh)

|  |
| --- |
| #!/bin/bash  # Demonstrate flow control in bash  # We're taking one arguement  key=$1  case $key in  Yes|yes)  echo "You said Yes or yes"  ;;  No|no)  echo "You said No or no"  ;;  \*)  echo "You said neither Yes/Yes nor No/no"  ;;  esac |

Comments:

* case $key in
  + This begins our case statement, specifying that we’ll be inspecting the value in $key
    - $key is the value of our parameter (first parameter) that we used when executing the script (key=$1)
  + Additional entries listed below (after “in”) will compare the value in $key to the value specified in the entry itself
* Yes|yes)
  + Our first pattern we’re trying match to.
  + Does $key contain “Yes” or “yes” (notice the lowercase vs uppercase)
  + ;; is equivalent to break; statements in other languages
    - Terminate execution of case
* No|no)
  + Our second pattern we’re trying to match to.
  + Does $key contain “No” or “no” (notice the lowercase vs uppercase)
  + ;; is equivalent to break
* \*)
  + Acts as default value and is executed when no other pattern match is found
* esac is always required to indicate the end of a case statement