I. Intro

1. A shell script contains commands that are executed as if you typed them in the terminal.

2. We'll be using Vim for this tutorial

a. Install Vim : sudo apt-get install vim

b. Vim Commands

1. i : insert mode

2. <ESC> : enter command mode

i. w : Save / Don't Exit

ii. wq : Save / Quit

iii. q! : Quit / Discard Changes

iv. w : Move to front of next word

v. b : Move backwards to front of word

vi. 0 : Move to start of line

vii. $ : Move to end of line

viii. G : Jump to last line

3. Move around with arrows

4. :set number : Displays line numbers

5. :syntax on : Syntax Highlighting

6. :set tabstop=2 : Spaces in tab

7. :set autoindent : Indent new lines

8. Save these in your home/~/.vimrc file

a. Find out what vimrc file you are using with this command in Vim :echo $MYVIMRC

To delete bunch the lines:

1: come out of insert mode by pressing ESC

2: press v

3: use arrows to select the rows

4: press d

To delete the entire vim

1: come out of insert mode by pressing ESC

2: press gg

3: press d

4: press G

To Copy lines and paste

1: come out of insert mode by pressing ESC

2: press v

3: use arrows to select the rows

4: press y

5: press p

3. Hello World Script

# The #! shebang tells the system the interpreter to use for the script

#!/bin/bash

# Comment

echo 'Hello World' # Print the string to the screen

a. To make it executable chmod 755 hello\_world

b. Execute with ./hello\_world

c. The numbers after chmod define who can do what with the file

d. The numbers represent the Owner, the Group and Everyone else

e. What the numbers mean

1. 7 : Read, Write & Execute

2. 6 : Read & Write

3. 5 : Read & Execute

4. 4 : Read Only

5. 3 : Write & Execute

6. 2 : Write Only

7. 1 : Execute Only

8. 0 : None

4. We define variables like this myName="Derek"

a. The variable name starts with a letter or \_ and then can also contain numbers

b. The shell treats all variables as strings

c. When declaring a variable you can't have whitespace on either side of the =

d.

#!/bin/bash

declare -r NUM1=5 # Declare a constant

num2=4

# Use arithmetic expansion for adding

num3=$((NUM1+num2))

num4=$((NUM1-num2))

num5=$((NUM1\*num2))

num6=$((NUM1/num2))

# Place variables in strings with $

echo "5 + 4 = $num3"

echo "5 - 4 = $num4"

echo "5 \* 4 = $num5"

echo "5 / 4 = $num6"

echo $(( 5\*\*2 ))

echo $(( 5%4 ))

# Assignment operators allow for shorthand arithmetic

# +=, -=, \*=, /=

rand=5

let rand+=4

echo "$rand"

# Shorthand increment and decrement

echo "rand++ = $(( rand++ ))"

echo "++rand = $(( ++rand ))"

echo "rand-- = $(( rand-- ))"

echo "--rand = $(( --rand ))"

# Use Python to add floats

num7=1.2

num8=3.4

num9=$(python -c "print $num7+$num8")

echo $num9

# You can print over multiple lines with a Here Script

# cat prints a file or any string past to it

cat << END

This text

prints on

many lines

END

II. Functions

1. You can use functions to avoid the need to write duplicate code

2. Delete all code in Vim with gg then dG

3. #!/bin/bash

# Define function

getDate() {

# Get current date and time

date

# Return returns an exit status number between 0 - 255

return

}

getDate

# This is a global variable

name="Derek"

# Local variable values aren't available outside of the function

demLocal() {

local name="Paul"

return

}

demLocal

echo "$name"

# A function that receives 2 values and prints a sum

getSum() {

# Attributes are retrieved by referring to $1, $2, etc.

local num3=$1

local num4=$2

# Sum values

local sum=$((num3+num4))

# Pass values back with echo

echo $sum

}

num1=5

num2=6

# You pass atributes by separating them with a space

# Surround function call with $() to get the return value

sum=$(getSum num1 num2)

echo "The sum is $sum"

III. Conditionals / Input

1.

#!/bin/bash

# You can use read to receive input which is stored in name

# The p option says that we want to prompt with a string

   read -p "What is your name? " name

   echo "Hello $name"

   read -p "How old are you? " age

   # You place your condition with in []

   # Include a space after [ and before ]

   # Integer Comparisons: eq, ne, le, lt, ge, gt

   if [ $age -ge 16 ]

   then

   echo "You can drive"

   # Check another condition

   elif [ $age -eq 15 ]

   then

   echo "You can drive next year"

   # Executed by default

else

  echo "You can't drive"

# Closes the if statement

fi

2. Extended integer test

#!/bin/bash

read -p "Enter a number : " num

if ((num == 10)); then

echo "Your number equals 10"

fi

if ((num > 10)); then

echo "It is greater then 10"

else

echo "It is less then 10"

fi

if (( ((num % 2)) == 0 )); then

echo " It is even"

fi

# You can use logical operators like &&, || and !

if (( ((num > 0)) && ((num < 11)) )); then

echo "$num is between 1 and 10"

fi

# && and || can be used as control structures

# Create a file and then if that worked open it in Vim

touch samp\_file && vim samp\_file

# If samp\_dir doesn't exist make it

[ -d samp\_dir ] || mkdir samp\_dir

# Delete file rm samp\_file

# Delete directory rmdir samp\_dir

3. Testing strings

#!/bin/bash

str1=""

str2="Sad"

str3="Happy"

# Test if a string is null

if [ "$str1" ]; then

echo "$str1 is not null"

fi

if [ -z "$str1" ]; then

echo "str1 has no value"

fi

# Check for equality

if [ "$str2" == "$str3" ]; then

echo "$str2 equals $str3"

elif [ "$str2" != "$str3" ]; then

echo "$str2 is not equal to $str3"

fi

if [ "$str2" > "$str3" ]; then

echo "$str2 is greater then $str3"

elif [ "$str2" < "$str3" ]; then

echo "$str2 is less then $str3"

fi

# Check the file test\_file1 and test\_file2

file1="./test\_file1"

file2="./test\_file2"

if [ -e "$file1" ]; then

echo "$file1 exists"

if [ -f "$file1" ]; then

echo "$file1 is a normal file"

fi

if [ -r "$file1" ]; then

echo "$file1 is readable"

fi

if [ -w "$file1" ]; then

echo "$file1 is writable"

fi

if [ -x "$file1" ]; then

echo "$file1 is executable"

fi

if [ -d "$file1" ]; then

echo "$file1 is a directory"

fi

if [ -L "$file1" ]; then

echo "$file1 is a symbolic link"

fi

if [ -p "$file1" ]; then

echo "$file1 is a named pipe"

fi

if [ -S "$file1" ]; then

echo "$file1 is a network socket"

fi

if [ -G "$file1" ]; then

echo "$file1 is owned by the group"

fi

if [ -O "$file1" ]; then

echo "$file1 is owned by the userid"

fi

fi

4. With extended test [[ ]] you can use Regular Expressions

#!/bin/bash

read -p "Validate Date : " date

pat="^[0-9]{8}$"

if [[ $date =~ $pat ]]; then

echo "$date is valid"

else

echo "$date is not valid"

fi

5. # Read multiple values

#!/bin/bash

read -p "Enter 2 Numbers to Sum : " num1 num2

sum=$((num1+num2))

echo "$num1 + $num2 = $sum"

# Hide the input with the s code

read -sp "Enter the Secret Code" secret

if [ "$secret" == "password" ]; then

echo "Enter"

else

echo "Wrong Password"

fi

6. You can set what separates the values with IFS

#!/bin/bash

# Store the original value of IFS

OIFS="$IFS"

# Set what separates the input values

IFS=","

read -p "Enter 2 numbers to add separated by a comma" num1 num2

# Use the parameter expansion ${} to substitute any whitespace

# with nothing

num1=${num1//[[:blank:]]/}

num2=${num2//[[:blank:]]/}

sum=$((num1+num2))

echo "$num1 + $num2 = $sum"

# Reset IFS to the original value

IFS="$OIFS"

# Parameter expansion allows you to do this

name="Derek"

echo "${name}'s Toy"

# The search and replace allows this

samp\_string="The dog climbed the tree"

echo "${samp\_string//dog/cat}"

# You can assign a default value if it doesn't exist

echo "I am ${name:-Derek}"

# This uses the default if it doesn't exist and assigns the value

# to the variable

echo "I am ${name:=Derek}"

echo $name

7. Use case to when it makes more sense then if

#!/bin/bash

read -p "How old are you : " age

# Check the value of age

case $age in

# Match numbers 0 - 4

[0-4])

echo "To young for school"

;; # Stop checking further

# Match only 5

5)

echo "Go to kindergarten"

;;

# Check 6 - 18

[6-9]|1[0-8])

grade=$((age-5))

echo "Go to grade $grade"

;;

# Default action

\*)

echo "You are to old for school"

;;

esac # End case

8. Ternary Operator performs different actions based on a condition

#!/bin/bash

can\_vote=0

age=18

((age>=18?(can\_vote=1):(can\_vote=0)))

echo "Can Vote : $can\_vote"

IV. Parameter Expansions and Strings

1. Strings

#!/bin/bash

rand\_str="A random string"

# Get string length

echo "String Length : ${#rand\_str}"

# Get string slice starting at index (0 index)

echo "${rand\_str:2}"

# Get string with starting and ending index

echo "${rand\_str:2:7}"

# Return whats left after A

echo "${rand\_str#\*A }"

V. Looping

1. While Loop

#!/bin/bash

num=1

while [ $num -le 10 ]; do

echo $num

num=$((num + 1))

done

2. Continue and Break

#!/bin/bash

num=1

while [ $num -le 20 ]; do

# Don't print evens

if (( ((num % 2)) == 0 )); then

num=$((num + 1))

continue

fi

# Jump out of the loop with break

if ((num >= 15)); then

break

fi

echo $num

num=$((num + 1))

done

3. Until loops until the loop is true

#!/bin/bash

num=1

until [ $num -gt 10 ]; do

echo $num

num=$((num + 1))

done

4. Use read and a loop to output file info

#!/bin/bash

   while read avg rbis hrs; do

   # printf allows you to use \n

   printf "Avg: ${avg}\nRBIs: ${rbis}\nHRs: ${hrs}\n"

   # Pipe data into the while loop

   done < barry\_bonds.txt

   5. There are many for loop options. Here is the C form.

   #!/bin/bash

   for (( i=0; i <= 10; i=i+1 )); do

   echo $i

   done

   6. We can cycle through ranges

   #!/bin/bash

   for i in {A..Z}; do

   echo $i

   done

   7.

VI. Arrays

1. Bash arrays can only have one dimension and indexes start at 0

2. Messing with arrays

#!/bin/bash

# Create an array

fav\_nums=(3.14 2.718 .57721 4.6692)

echo "Pi : ${fav\_nums[0]}"

# Add value to array

fav\_nums[4]=1.618

echo "GR : ${fav\_nums[4]}"

# Add group of values to array

fav\_nums+=(1 7)

# Output all array values

for i in ${fav\_nums[\*]}; do

echo $i;

done

# Output indexes

for i in ${!fav\_nums[@]}; do

echo $i;

done

# Get number of items in array

echo "Array Length : ${#fav\_nums[@]}"

# Get length of array element

echo "Index 3 length : ${#fav\_nums[3]}"

# Sort an array

sorted\_nums=($(for i in "${fav\_nums[@]}"; do

echo $i;

done | sort))

for i in ${sorted\_nums[\*]}; do

echo $i;

done

# Delete array element

unset 'sorted\_nums[1]'

# Delete Array

unset sorted\_nums

VII. Positional Parameters

1. Positional parameters are variables that can store data on the command line in variable names 0 - 9

a. $0 always contains the path to the executed script

b. You can access names past 9 by using parameter expansion like this ${10}

2. Add all numbers on the command line

#!/bin/bash

# Print the first argument

echo "1st Argument : $1"

sum=0

# $# tells you the number of arguments

while [[ $# -gt 0 ]]; do

# Get the first argument

num=$1

sum=$((sum + num))

# shift moves the value of $2 into $1 until none are left

# The value of $# decrements as well

shift

done

echo "Sum : $sum"

SORT Command

nohup mode

awk

sort - sort file

sort -t, -k1 file1 - sort file1 by column 1, delimited by comma (,)

sort -r -t, -k1 file1 - in reverse

sort -n -t, -k1 file1 - sort by numerical value

sort -f file1 - case insensitive sort

sort -t'|' +0 -2 - sort by the first 2 fields, starting from 0+1th field and end at 2nd field.

sed - special editor

sed 's/pattern/replacement/g data.txt search and replacement globally

sed 's/pattern/replacement/gi' - data.txt search and replacement globally and case insensitively

sed 's/pattern/&replacement/gi' data.txt - & represents the pattern found

sed 's/\(group\)pattern/\1/gi' data.txt - \1 contains value of 1st group

sed 's/hello/bye/2g' data.txt - replacements second occurrence of hello

sed 's/hElLo/bye/2gi’ data.txt - replacements second occurrence of hello case insensitive

sed '/pattern/p' data.txt - will print out occurrences of pattern

sed -e 'command1' -e 'command2' data.txt - run multiple commands

VAR="McDonald"

sed 's/'"$VAR"'/replacement/' data.txt - using external variables in sed

sed '2 s/pattern/replacement/' data.txt - replace only for line 2. you can also specify line range, 1,10

sed '/pattern1/ s/pattern2/replacement/' data.txt - find pattern1, for those lines, replace pattern2 with replacement

sed '/start/,/end/ s/pattern/replacement/' data.txt - using patterns as line number range. you can also mix line number with pattern

sed ‘3,/end/ s/pattern/replacement/' data.txt - using patterns as line number range. you can also mix line number with pattern

sed '1,10d' data.txt - deletes line 1 to 10. you can also mix pattern range as well

sed -i ‘1,10d' data.txt - goes into the file and deletes line 1 to 10. you can also mix pattern range as well

sed '100,$d' data.txt - delete from line 100 to end of file

sed '100,$!d' data.txt - delete from line 1 to line 100. ! means inverse operation.

sed 'y/ABC/abc/' data.txt - replacement each character with corresponding characters

<http://www.grymoire.com/Unix/Sed.html>

-F is for delimiter

awk -F'|' 'BEGIN{ initialization } { code block } END{ finalization }' filename- general syntax. code block will be executed for every line in the file

default delimiter is space ' '

awk -F, '{ print $0 }' data.txt - $0 represents the whole line of the file

awk -F, '{ print $1 }' data.txt - $1 represents the first column of the file

awk -F, '{ print $1$2 }' data.txt - prints firstsecond column

awk -F, '{ print $1,$2 }' data.txt - prints first second column

awk -F, '{ OFS="|";print $1,$2 }' data.txt - OFS is output field delimiter. prints first|second column

awk -F, '{ print $1"|"$2 }' data.txt - prints first|second column

awk -F, 'NR>1 { OFS="|";print $1,$2 }' data.txt - prints first|second column for rows greater than 1

awk -F, '/2013-05/ { OFS="|";print $1,$2 }' data.txt - prints first|second column for rows that matches 2013-05

awk -F, '/2013-05/ { OFS="|";print $1,$2,NF }' data.txt - prints first|second|no of fields columns for rows that matches 2013-05

awk -F, '/2013-05/ { OFS="|";print $1,$2,FILENAME }' data.txt - prints first|second|filename columns for rows that matches 2013-05

awk -F, '/2013-05/ { OFS="|";print $1,$2,”’$HOME'"}’ data.txt - prints first|second|env\_variable columns for rows that matches 2013-05

awk -F, { OFS=“|”;if($1~/2013-05/){print $1,$2}}’ data.txt - prints first|second|env\_variable columns if first column matches 2013-05

awk ‘BEGIN { FS=“,”; sum = 0; count = 0} \

{ if(NR>=2) { sum = sum+$5; count++; \

printf(“Day:%d Avg:%f Sum:%f field:%f \n”,count,sum/count,sum,$5) } }’ data.txt

VAR="value" - define a variable

print VAR - print a variable

NF is number of fields in each line

$NF is value of last field

print "'$EXTERNAL\_VAR'" - double single $EXTERNAL\_VAR single double

VAR~/pattern/ - pattern patching

VAR!~/pattern/ - does not match

FS returns value of field separator, same as -F

OFS returns output field separator

print $1,$2 is separated

NR returns line number

FILENAME returns current filename

awk -v var1=value1 - pre-assign values to variables in awk

awk -F, 'conditions { code block }' filename - condition must be right before code block

cut - get columns from file

cut -d, -f1-2 filename - print 1st to 2nd columns

cut -c1-5 filename - print first 5 characters

cut -d, -f2- filename - print from 2nd column to last column

cut -d, -f1,4,5 filename - print 1st, 4th and 5th column

cut -d, -f-3 filename - print from 1st to 3rd column

cut -d, -f1,3-5 filename - print 1st, 3rd to 5th column

cut -d, -f1,2,3 data.txt

cut -d, -f3-5 data.txt

cut -d, -f-5 data.txt

cut -d, -f5- data.txt

cut -d, -f1,5- data.txt

cut -c1-10 data.txt

cut -c1,10-20 data.txt

wc - word count

wc -l - line count

wc -c - print number of bytes

wc - print number of newline, word and bytes

wc -m - character count

wc -w - word count

wc -l < data.txt gives the line count

wc -l data.txt - gives the line count and filename

wc data.txt - gives no: of lines, words, bytes, filename

wc -c data.txt - gives no: of bytes

wc -w data.txt - gives no: of words

wc -m data.txt - gives no: of characters.

last - check login history

last -10 - show last 10 logins

last -F - show full login information