Shell Scripting

Why shell scripting?

- Saves time and effort
- Serves as a form of documentation on how to perform specific tasks
- Can be scheduled if used in Cron job
- Performs complex tasks when using advanced techniques, like looping and decision making
- Can be used to carry out tasks unattended. For example, a shell script can monitor the system while the system administrator is out of office.
- Can be used to collect important information like application logs
- Many other uses that will be demonstrated in this section

Which shell do you mean?

- Different Linux/Unix systems have different default shells. The sh (Bourne Shell) was one the dominant one before BASH (Bourne Again Shell) was introduced.
- Other types of shells include ksh (Korn shell), and csh (C-shell). The default Red Hat and Ubuntu shell is the BASH.
- Choosing a shell is a matter of taste, as much as a matter of task requirements. For example, csh used to make C programmers feel *at home*.
- Changing the current shell is as easy as typing `ksh` to start using the Korn shell from a BASH session.
- Shells – in general – provide a way to group multiple commands together in a single or multiple files. They are easy to install because they do not require any additional setup.
- If the task at hand is much more complex than to be solved by a bunch of shell commands, you can opt to use a fully fledged language like Python or Perl.

Working with the CLI using emacs

- Interacting with the Command Line Interface can be tailored to your convenience.
- If you are an emacs user, you can use the following shortcuts to communicate with the shell:
  - CTRL-E: jump to the end of the current line
  - CTRL-A: move to the line beginning
  - CTRL-P: brings the last used command. Press it again to bring the second last and so on.
  - CTRL-R: lets you search for commands in *history* of the last used ones.

Vi users, you're welcome too!

- If you prefer vi to emacs, you can use it with the CLI, just type `set -o vi` to enter vi mode.
- In vi, you have to press `ESC` to enter command mode, where you can enter navigation commands. When you want to enter text, press `i`. You can use the following commands to interact with the CLI:
  - $: go to the end of the current line
  - 0: jump to the start of the current line
  - k: brings the last command typed. Pressing it again brings the second last and so on.
  - /: searches the history for commands
- The above is just the most common vi commands that can be used. You can work with other commands like w, b for moving by word for example.
- You can return back to emacs mode using `set -o emacs`

The process file descriptors (fd)

- In UNIX paradigm, a process communicates with the system using *channels* called *file descriptors (fd)*.
- At least 3 channels are available to a given process:
  - Standard Input (STDIN), from which the process accepts input. For example, the shell process typical accepts input from the keyboard. It is numbered as 0.
  - Standard Output (STDOUT), to which the process directs any output produced. It is numbered as 1.
  - Standard Error (STDERR), to which any error messages are directed. It is numbered as 2.
- You can always examine the current fd's used by a process by listing the files under the virtual directory /proc. For example ls –l /proc/45163/fd, where 45163 is the process number.

Communicating with fd's

- Redirect standard output and standard error to a file using > or >> (remember error fd is 2)
- The > sign clears the file before writing to it, while the >> sign appends to the file.
- Direct both input and output to the same destination using &>
- Instruct the process to take it's standard input from a file using <
- Inject the STDOUT of a command to the STDIN of another using the | (pipe)

Using variables

- A variable is a container of value.
- It is declared by assigning a value to it. For example, `USER=joe`.
- There must not be any spaces between the = sign, the variable name, and the assigned value.
- Values should be enclosed in single quotes, especially if they contain spaces.
- To call a variable, you prefix it with a $ sign. You can also add curly braces like so: `${VAR}`. This is recommended if you are using the variable with literal text. For example, the path is `${ORACLE_HOME}/oraInventory`.
- Enclosing the variable name in single quotes will prevent the variable from being resolved to it's value. For example, '$HOME' will print $HOME, while "$HOME" will print the home directory path of the current user.
- You can add your *environment* variables in `~/.bash_profile` or `~/.profile`

The `cut` command

- This is our first example of *filter commands*, those which take STDIN, perform some processing on it before returning the result as STDOUT.
- The `cut` command: it is used to display parts of the input text. It is often used with delimited files like csv, and /etc/passwd.
- The default delimiter is the TAb character, but this can be changed using the –d command line argument
- To select the fields you want to display, use the –f option followed by a comma separated list of numbers. For example: `cut -d , -f 1,3 < sample.csv` will display fields 1 and 3, which are separated by a comma in a typical csv file, sample.csv
- The < sign is redundant here because `cut` accepts a file as it's first argument. It is used in the example for demonstration purposes.

The `sort` command

- Sorts the lines of text given to it. You'll have to provide the key column on which the sort operation is done, and the *type* of sort: numerical or dictionary based.
- The typical usage is as follows: `sort -t, -k1,1 -n < sample.csv`, which will sort the lines of sample.csv in an integer based way, regarding the comma as a column delimiter, and the key column is 1. The key column can be more than one so we must specify that it will take only one column by inserting a comma and providing an *end* value: `k1,1`.
- Use –d to make it work in dictionary mode
- We can use –r to reverse the order, -u to print unique values only, and -b to ignore leading spaces.
- Sort is usually combined with another filter command like `cut` or `uniq`

The `uniq` command

- In it’s simplest form, it returns the unique lines of a given text input. But you have to use the –u switch or it won’t process the input at all.
- It has some more complex usages such as:
  - -d : to print duplicated lines
  - -c : to count the number of duplicated/non-duplicated lines
- One prerequisite for `uniq` is that the lines must be sorted first before being input. So if they are not sorted by default, the `sort` command is used.
- Because of the above requirement, `uniq` usually takes the standard output of `sort`, rather than the raw file. For example,

  ```
  sort -t, -k1,1 sample.csv | uniq -c
  ```

  which will print the number of times a line has been entered in the sample.csv file.

The `wc`, and `tee` commands

- The `wc` (word count) command is used to count words, lines, and characters in the text given to it.
- If run without any options, it will output the three values. But more often than not you will use one of it's switches:
  - `-l` for lines
  - `-w` for words
  - `-c` for characters
- The `tee` command is used to *view* the standard output of command while it's being written to a file. For example,
  ```
  find / -name *.log | tee output.txt
  ```
  will print the output of the find command both to the screen and to output.txt file at the same time.

The `head` and `tail` commands

- The `head` command prints the first 10 lines of a file. It has the following useful options:
  - --*lines=n*: will print only the specified number of lines. It can be written as -n
    For example `head -5 sample.csv`
  - `--lines=-n`: will print all lines in the file except the last n lines.
- The `tail` command prints the last 10 lines of a file. It has the following useful options:
  - --lines=n : prints the last n number of lines in a file. It can be used as tail –n
    For example `tail -5 sample.csv`
  - `--lines=+n`: prints all lines in a file starting with line number n
  - -f : will keep the file open, displaying any new lines added to it. This is typically used with log files.
  - `--pid=PID`: makes the command exit after the process which is writing to the file you're tailing exits. Useful so that you know if the process has exited or it's just not writing any output at the current moment.

The `grep` command

- Searches for text in it’s standard input and outputs it. You can use *regular expressions* in pattern matching.
- It has the following number of useful options:
  - -c: get the number of matches instead of printing them.
  - -i: ignore case when searching.
  - -v *text*: matches if the text is *not* present
  - -l: print the file names containing the matched text
  - -R: perform a recursive search. That is, in the current directory and all subdirectories

Basics of a BASH script

- All the commands that you write in BASH shell can be put in a file called a shell script to automate them.
- You can comment any line of code in a BASH script by adding a # at the start.
- Multiples commands can be added to the same line by separating them with semicolons (;)
- If you want the script to be *self executable*, you must add the interpreter at the first line of the script. For example `#!/bin/bash`. This is referred to as *shebang*. Otherwise, the script will only run if you preceded the file with the interpreter. For example, `sh script.sh` or `bash script.sh`
- The file extension has no effect on execution. Adding .sh to the end of scripts is a good practice, though, to easily identify a file without having to open it.

Interacting with the user

- The `echo` and `printf` commands are used to print text to the screen.
- Both command are similar but `printf` allows you to output special characters like tabs (\t). You also manually add the newline character (\n).
- The echo command has the following useful arguments:
  - -e : make it work like printf (interpret special characters like \t and \n)
  - -n : do not add newline character to the end of the line.
- You can accept input from the user using the `read` command, followed by a variable in which the text will be saved. For example: `read user_input`

Script arguments

- You can add command line arguments to the script to be used inside it.
- They are internally interpreted by BASH as $1, $2, for the first and second arguments. $3 and $4 for the third and fourth an so on.
- The following special variables can be useful in your script:
  - $0 holds the name of the script file
  - $# contains the number of arguments passed to the script
  - $* contains all the arguments passed at one
- The command line argument variables with the special ones can be used to test whether or not the user has supplied the correct input if at all, and print a friendly usage message accordingly.

Shell Functions

- They can be used the same way functions are used in any programming language; yet they are don't use parentheses.
- The syntax is `function` *`function_name`* `{` *`code`* `}` and it is called like this: `function_name`
- A function executes a block of code when called. It is useful in centralizing application logic. For example, printing usage or error messages can be put in a function to avoid repeating code.
- Usage is not restricted to shell scripting; you can also use them as an *alias* to your shell commands. For example, you can add a function your `.bash_profile` and use it as a command.

Variable visibility

- Variables are global by default. That is, they can be accessed anywhere in the code. But a function can make variable accessibility limited to their own scope by using the keyword *local*.
- For example: `$a` if a global variable, by using `local a` makes it only accessible within the function limits.
- Using local variables enables functions to do any required processing on a global variable without the risk of changing it's value globally.
- The variable retains its original value as soon as code execution leaves the function where the variable was created as *local*.

Decision making

- You can make decisions in your code by using the *if* statements just like you use it in any programming language.
- The *if* code block starts with *if* followed by square brackets containing the condition to be tested (with spaces before and after the square brackets). After which you type *then* (either on a separate line or preceded by a semicolon). Then you type the code that will run if the test is truthful. You can add *elif* and *else* statements to provide alternate decisions if the main test fails. An *if* block must be ended by *fi*.
- The following is an example:

```
id=`id -u`
if [[ $id -eq 0 ]]; then
     echo "This script cannot be run as root"
else
    echo "Welcome to our script"
fi
```

The *test* command

- The [and] used with the if conditional statement is actually a *shortcut* to the `/bin/test` command, which is used to examine a logical condition.
- The following is some of the most commonly used operators with `test`. The full list can be found by typing `man test`:

Type	Operand	Sign	Example
Equal to	String	=	`$user = 'Sam'`
	Number	-eq	`$id -eq 1`
Not equal to	String	!=	`$user != 'John'`
	Number	-ne	`$id -ne 5000`
Greater than	Number	-gt	`$count -gt 0`
Less than	Number	-lt	`$limit -lt 100`
Greater than or equal	Number	-gte	`$count -gte 0`
Less than or equal	Number	-lte	`$limit -lte 100`

Testing for file attributes

- The following is the most common operators used to test against file attributes. The full list can be found at `man test`:

Sign	Usage
-d *directory*	Directory exists
-e *file*	File exists
-f *file*	File exists and is a regular file (not a block device for example)
-r *file*	File is readable
-s *file*	File is not empty
-w *file*	File is writable

The *case* statement

- Used to test against several possible values, just like `elif` but more concise.
- The syntax is as follows:

```
case $variable in
 value 1) code ;;
 value 2) code ;;
 value 3) code ;;
 value x) code ;;
 *) default code when all the above fails
esac
```

- A typical usage for this type of conditional is when you are expecting an input from the user, like yes, or y, to respond accordingly.

Repetitive tasks (looping)

- BASH provides the `for..in` loop to iterate over a group of values, and perform some commands on each one.
- It can also accept a group of files by using *globing*, which means using an asterisk * or a question mark `?` to find files by parts of their names. For example:

  ```
  for f in *.log; do
   gzip $f
  done
  ```
- The loop starts with the keyword for followed by the iterator (the variable to which each value will get assigned), then a list of values or a globed file name. Then a semicolon (;) and the `do` keyword are added. After writing the code for the loop, you have to close it by adding `done` to it's end.
- A loop can also work on a list of values separated by a space like:

  ```
  for i in user1 user2 user3; do
   echo $i
  done
  ```

The `for` loop

- The classic for loop is available in BASH with some slight modification.
- The syntax is as follows:

```
for (( i=0; i<$count; i++)); do
 echo $i
done
```

- The loop starts with a for keyword, followed by double brackets, in which the loop parameters are entered:
  - The iterator (i)
  - The condition that makes the loop continue as longs as it's true ($i is less than $count)
  - The statement that updates the iterator (i++)
- Notice that i is not preceded by a $ in the loop definition.
- The loop body is terminated by the `done` keyword

The `while` loop

- The while loop will continue to iterate as long as a specific condition is met.
- It is often used in reading strings such as user input or a text file.
- The syntax of a while loop is as follows:

```
while [[ condition ]]; do
 code
done
```

- Sometimes you need to make infinite loops. This is typically done in daemons that work in the background, listening for an event and acting accordingly. In such a case, you can use a while loop as follows:

```
while true; do
 code
 sleep n seconds
done
```

- In the above example, you don't care about the loop condition because you want it to work "continuously". But it's a good practice to put a *sleep* statement at the end of each iteration so that you don't exhaust your machine resources.

Perl

- It's a fully fledged programming language that is must more powerful than BASH.
- Using Perl or Python to do more complex tasks is highly recommended than using languages such as C, C++, or Java; because it is more suited from shell scripting.
- In Perl you can do lots of things in very few lines of code that are easy to debug than the other mentioned languages.
- Choosing between Perl and Python is a personal preference; as both of the languages – roughly – offer the same power.
- Perl may be a little less readable than Python

Basics of a Perl script

- It comes preinstalled with most Linux distribution under `/usr/bin/perl`
- You can install it by typing `yum install perl` for Red Hat, or `apt-get install perl` for Debian
- Statements are terminated by semicolons
- Comments are started by a hash
- Blocks of code are enclosed in curly braces { }
- Perl statements cannot be used outside a Perl script.
- There are lots of similarities between Perl and BASH. This makes it easier for a BASH coder to learn Perl

Perl arrays

- In Perl, numbers and strings are called scalar values. They are denoted by the $ sign.
- Arrays are groups of scalar values stored in one variable. They are enclosed in brackets, and separated by commas. Arrays are best dealt with using loops. Perl recognizes an array by the @ sign.
- @ARGV is a special type of Perl array that contains the command arguments used when calling the script (compare it to $* in BASH)
- Arrays are zero based. That is, the first item is at location 0, the second is at 1 and so on
- You can get the number of items in an array by using `$#arrayname + 1`
- You can obtain a value from an array using its index like this `$arrayname[1]`
- Arrays can be sliced, which means taking only a subset of the array members by using this form `@array2 = @array1[x,y]`

Perl hashes

- The are the same as arrays but they have *keys*. Those keys are used to call the values instead by them instead of by their index.
- Hashes are denoted by % sign
- They are created as `%hash = ('key',value)` or, for better readiblity, `%hash = ('key' => value)`
- Another form of creating hashes is `%hash('key') = value`
- You can call hashes the same way you call arrays but specify the key instead of index, and curly braces instead of brackets, like this `$hash{key}`
- Slicing in hash is done the same way as arrays: `@output=@hash{key_x,key_y}`

Working with Perl

- You can print text tot the screen by using the `print` function.
- The `print` function interprets special characters like \n and \t by default.
- The . (dot) character is used to concatenate strings
- Input from the user is accepted by using assigning a variable to the `<STDIN>`, and using the `chomp` command to strip any trailing spaces
- You can accept multiple lines from the user if you discarded the `chomp` function and used an array to accept user input instead of a scalar variable. In this case, different lines will form different items of the array.
- In case of multiple entries, the user can finish input by pressing the EOF character (typically CTRL-D)
- You can execute shell commands and store their output in the following form: `$`*`var`*`=`` ` ``*`command`*`` ` ``

Perl script arguments

- They are stored in the built-in array `@ARGV`
- The first argument has the place 0 (not 1 as in BASH)
- The script name is provided by the special variable $0
- If no arguments are supplied, the `$#ARGV` (items count) will be -1 rather than 0
- You can accept input from standard input by assigning `<STDIN>` to the `@ARGV` array

Perl if conditional

- It is used for decision making in the script
- The syntax is as follows:

```
if (condition) {
 code if condition is true
}
elseif (condition) {
 code if above condition is false
}
else {
 code if all conditions are false
}
```

Perl operators

Type	Operand	Sign	Example
Equal to	String	eq	`$user eq 'Sam'`
	Number	==	`$id == 1`
Not equal to	String	ne	`$user ne 'John'`
	Number	!=	`$id != 5000`
Greater than	Number	gt	`$count gt 0`
Less than	Number	lt	`$limit lt 100`
Greater than or equal	Number	gte	`$count gte 0`
Less than or equal	Number	lte	`$limit lte 100`

Perl loops

- A for loop is used to execute code for a specific number of times. It has the following syntax:

```
for ($i = 0; $i<10; i++){
 code
}
```

- The foreach loop works best with arrays. The syntax is as follows:

```
foreach $var(@:variables){
 $var holds the current array item value
}
```

- The while loop is used when you want code to execute as long as a condition is met. The syntax is as follows:

```
while (condition){
 code
}
```

- Infinite loops can be done like this:

```
while (true) {
 code
 sleep n seconds
}
```

Python scripting

- Python is generally more readable than Perl. Some argue that it is easier to learn as well
- Most UNIX/Linux distributions have Python already installed
- The language is currently maintained in two streams: 2 (the latest now is 2.7), and 3 (the latest now is 3.5). Both of which are considerably different than each other in syntax and other language aspects.
- It has it’s own interactive shell, which is useful to test your commands before putting them in a script
- We are going to deal with Python 2 in this course because it’s still the default Python version in Linux

Basics of a Python script

- If not already installed, Python can be installed using yum, or apt-get. If you want to install Python 3.5, you can download it the tar.gz file and compile it.
- Multiples versions of Python can coexists on the same machine, with the default one called Python, and the other one called Python3.x
- The commands are written on separate lines
- Blocks of code are identified by indentation (no curly braces here). Python typically required four space indentations to define a code block
- Comments are written preceded by the # sign

Python modules

- Python is said to have *batteries included*, which means that it has got a lot of built-in functionality that make it independently robust enough to handle most tasks.
- It uses *modules* to encapsulate different types of tasks. For example, the os module for dealing with the underlying operating system, the `time` module for providing date and time functions and so on
- The sys module is the one that we are going to cover in this course.

Working with Python

- It uses the print command to print text to the screen. The command is aware of special characters like \n and \t.
- It uses `raw_input()` function to accept input from the user.
- The + sign is used to concatenate strings, as well as performing normal addition function, depending on the type of operand used.
- Variables are declared like this: `var = value`

Python if conditional

- It has the following syntax:

```
if condition:
    code if true
elif condition:
    code if previous condition is false and this one is true
else:
    code if all conditions are false
```

- Notice the colon (:) after the if statement. This is how a block of code is started.
- Notice the all the code under if, elif, and else must be indented 4 spaces from the left. This is how Python knows where the block starts and ends.

Python lists, tuples, and dictionaries

- Lists are like arrays: a group of values saved assigned to one variable.
- They are created as follows: `listname = [val1, val2, val3]`
- Tuples are the same as lists, but they cannot be modified. They are created as follows: `tuplename = (val1,val2,val3)`
- You can call an item in a list, or a tuple like this `varname[`*`index`*`]`
- Dictionaries can be thought of as associative arrays. They contain key value pairs of values. You can call each item by its name instead of index.
- The are created as follows:
  `mydict = {key1:value1,key2:value2,key3,value3}`
- A dictionary item can be called like this `mydict[key2]`

Python loops

- Python has two kinds of loops: the for loop and the while loop
- The for loop has the following syntax:

```
for item in item_sequence:
    code
```

- The item_sequence could be any variable that can be iterated over. A list or a tuple are examples
- The while loop will continue execution until a condition fails. The syntax is as follows:

```
while (condition):
    code
```

- If you want to make an infinite loop, you'd have to first import the time module first so that you can pause the script before the next iteration. The syntax would be like this:

```
import time
while (True):
 code
 time.sleep(n seconds)
```

Python interaction with the OS

- Python provides `sys.argv`, from the sys module, which provides a list containing the command arguments passed
- The script name is stored in sys.argv[0], so the argument list starts at place 1.
- You can make Python execute shell commands by using the `subprocess` module. This module provides the call function which will execute the command you pass to it as a parameter.

Conclusion

- Shell scripting is a robust way to automate repetitive tasks
- BASH and Korn shells use the normal command lines instructions in their scripts
- For more complex tasks, you use a fully fledged language like Python or Perl
- Both languages does the same tasks as the normal shell providing more features
- Choosing to code using the shell commands or moving to a language like Perl or Python largely depends on the task complexity and you personal preference and comfort