Shell Scripting – Part I

Introduction to Scripting

- Scripting Languages
  - Unlike c, c++, java programs, scripting languages have no compiling
  - Code is converted to machine language line by line while executing
  - These are therefore ‘interpreted’ languages (v.s. ‘compiled’ languages)
  - Python, Perl, PHP, and Shell scripting are examples
  - For example to run a python script you would say:
    - python myscript.py
    - Let’s take a moment to discuss this

Introduction to Scripting

- Scripting Languages
  - The next features are all debatable and subjective, and the lines between scripting and programming are blurry
    - Usually smaller programs than compiled languages
      - Although even some games are written in scripting languages
    - Usually for a special run-time environment that automates the execution of tasks
      - The tasks could alternatively be executed one-by-one by a human operator
    - Usually much high level
    - Usually do not offer low-level features such as pointers
  - Python is a counter-example, that is a scripting language, with robust features (including graphics, machine learning, and other big libraries, no pointers, though)

Introduction to Scripting

- Script Files
  - Script filenames have an extension to indicate which scripting language is used
    - e.g. Python is .py, Perl is .pl, and Shell is .sh
  - As always, the extension is not inherently required
    - Though, it allows other programs like text editors to know and offer syntax highlighting, etc.
  - Recall, that to run a script file, it must be passed through its interpreter:
    - python mypycode.py
    - perl myplcode.pl

Introduction to Scripting

- Script Files
  - Wouldn’t it be nice, if your O.S. automatically knew which interpreter to run
    - Should be able to just type the filename
    - If it’s a python file, just run it through the python interpreter
  - Linux and Unix-like systems have this feature
  - It’s called ‘hash-bang’ (#!)
  - Specify as the first line of your script, which interpreter (the complete absolute path to it) you want to use

• if you want python	#!/bin/python
• If you want perl	#!/usr/bin/perl
• If you want bash	#!/bin/bash

Introduction to Shell Scripting

- Shell scripting is, quite simply:
  - Take all the shell commands you want to execute, and put them in order one after the other in a script file
- Recall, we use bash, but you could write a shell script for ksh, or cshell, or any other shell you are using
  - The hash-bang (#!) will indicate which shell to use
    - #!/bin/csh
    - #!/usr/bin/ksh
- Additionally, most shells (including bash) also offer if-then-else, loops, variables, functions, file i/o, etc.

Introduction to Bash Scripting

- Writing our first shell script, we know the file extension (.sh)
  - Create and open in your favoritest text editor (vim) a file called *myscript.sh*
- Since we are using bash, we know the #! line:
  - #!/bin/bash
- Lest the world collapse, let’s start with Hello World
  - echo “Hello World”

Writing our first Bash Script

- Here’s how *myscript.sh* looks

```
#!/bin/bash
echo “Hello World”
```

- *echo* is nothing special
- It’s a bash command that prints a string to the screen
- Try it in the terminal if you want

Running our First Bash Script

- Since we were diligent, and placed our #! line, we are promised that we can just type the filename, and run our script
- Kind of like how we run firefox, or gedit, or xeyes, or ls, or top, or,… you get the point
- So, save the script file, and exit back to the terminal
- Let’s type
  - myscript.sh

Running our First Bash Script

- Appreciate the error the previous step threw
- So, what happened?
  - Typing firefox, xeyes, ls, top, etc. work because these programs are 'installed' on your system
  - Installation simply means, their executable file, or a symbolic link to their executable file is placed in one of the following directories (you can configure more)
    - /bin/
    - /usr/local/bin/
    - /sbin/
    - /usr/local/sbin/
  - Executing any command in the terminal, means searching these directories for a file with that name, and executing that program

Running our First Bash Script

- Since we did not put our script file in one of those default directories, *myscript.sh* is not a recognized command
- To run, we must specify the path to our script file
- If you are in the same directory as the script
  - *./myscript.sh*
- Recall, . means ‘this directory’

Running our First Bash Script

- Appreciate the error thrown at you

Side Note on File Permissions

- All files in Linux have permissions assigned
- The permissions could be read (r), write (w), or execute (x)
- A file could be read only, write only, execute only, or a combination of any of these
- Additionally, Linux allows separate permissions for three types of entities – owner, group, world
- So a file could be allowed rwx for owner, rw for group Nirvana, and r only for rest of the world

Side Note on File Permissions

- Do *ls -al* to view the permissions of the files
- The first character indicates directory (d) or regular file (-)
- Notice three sets of rwx
- The first three characters are permissions for owner, then permissions for group, followed by permissions for rest of the world
- Any permission not allowed for a certain entity is denoted by a -

Side Note on File Permissions

- Notice our *myscript.sh* file has
    - rw-r--r--
  - Owner (you) has read and write permissions, while group and world have read only permission
  - These are the default permissions for a file
  - No one has execute (x) permission
  - No wonder we couldn’t execute the script
- We need to change the permissions of this file
- At least give owner execute (x) permission

Side Note on File Permissions

- The command to modify permissions of a file is *chmod*
- The syntax is:
  - chmod *new_permissions* filename
- You can only modify permissions of files you own
- Or if you are sudo, you can modify permissions of any file

Side Note on File Permissions

- Syntax of *new_permissions* in chmod
  - u=rwx,g=rx,o=r
- Thus to give owner (user) rwx, and group and others read only
  - chmod u=rwx,g=r,o=r myscript.sh
- This is tedious
- There is a shorter way of specifying *new_permissions*

Side Note on File Permissions

- Alternate (more useful) syntax of *new_permissions* in chmod
    - chmod *xxx* myscript.sh
  - Where xxx is a three digit number
  - The first digit is permission for owner, followed by digit for group, and then digit for others
  - Each digit is calculated as follows
    - r = 4, w = 2, x = 1, 0 = no permission
  - To give a combination of permissions, add the numbers

Side Note on File Permissions

- Each digit is calculated as follows
  - r = 4, w = 2, x = 1, 0 = no permission
- To give a combination of permissions, add the numbers for that entity
- e.g. To give user rwx, and all others r only
  - ***chmod 744 myscript.sh***
- Once changed, the permission will stay for the life of that file
- Try this and do *ls -al* to confirm

Running the Bash Script

- Finally!
- Run the script (from the same directory)
  - ./myscript.sh

Bash Script – Practice

- Write a bash script called *helpful.sh*
- On being run it should do the following automatically:
  - Make a directory ~/Documents/AutoDir
  - Fetch a file from the internet url home.manhattan.edu/~kqazi01/main.c into that directory
  - Create a symbolic link for that file called m.c
  - Display the number of lines in that file
- **Remember: Any thing you can run in the terminal, you can put as a line in the script, and vice versa!**