Beginner’s Guide to Bash

1 Goals for this session

This session is aimed squarely at beginners in the world of Bash scripting and Unix in general. If you don't know what Bash is, or if you've just used other people's Bash scripts without really understanding them, or you've tried to use Bash and ended up confused by its idiosyncracies then this session is probably for you.

By the end of it you will hopefully:

- know roughly what Bash is
- know where to read the docs and be aware of the some of the odd naming conventions in shell-land
- know how to automate Bash's behaviour by writing scripts
- understand when it's appropriate to use shell scripting
- understand the limitiations of shell scripting
- know where to find documentation and how to use it about shell scripting

2 What is Bash?

Bash is an example of family of programs, known as "shells" which take commands from the user and execute them. Bash is probably the most common shell and is usually the default on modern OSs that need one.

It's main job is to execute other programs on your behalf, and this follows the Unix philosphy of having small programs that do one thing well, and then giving you the ability to glue these programs together to achieve larger tasks.

For example, the shell will provide you with the ability to 'pipe' the output of one command into the input of another.

For example, this line will use the `wc` program to count the number of words in each file in the current directory, and then pipe this to the `sort`:

```
wc -w * | sort -n
```

Most commands you run, even the most basic ones, are external programs to which the shell is able to delegate, for example `which ls` will show you where the binary that shows you a list of files is stored.

3 Getting help

Most Unix-like systems, incuding OSX, come with a set of documentation files call man pages, which document in great detail everything you could ever want to know about a command on the system. They can be accessed by typing `man` followed by the name of the command you want to learn about, and the best place to start is perhaps the man page about the man command itself, and you can search man pages with the `apropos` command:

```
$ man man
MAN(1)                 Manual pager utils              MAN(1)

NAME
       man - an interface to the on-line reference manuals
[...]
$ apropos compress
archive_read_filter (3) - functions for reading streaming archives
archive_write_filter (3) - (unknown subject)
gzip (1)             - compress or expand files
[...]
```

4 Exercise One

Compress all of the files in a directory that don't match the `*.gz` suffix.

In the `test-data` directory of the repo will find a number of files, some of which are compressed, and some of which aren't.

The command to use for the compression is `gzip`.

Bash (and shells in general) make the general assumption that what you are trying to do is execute system commands, and so that is it's default interpretation of what you want to do, with everything else as just glue that sticks those elements together.

Naturally, this is what makes Bash well suited to automating system administration tasks, as these can usually be achieved with very little extra code: you can just run your tasks immediately, as you would type them without any preamble or configuration.

For example, one way we might solve this exercise in Scala would be a program like this:

```
import java.io.File

def process(f: File, indent: Int = 0): Unit = {
  if(f.isDirectory)
    for(child <- f.listFiles) process(child)
  else if(!f.getName.endsWith(".gz"))
    /* ... either call out to `gzip` or use the JDK library... */
    Runtime.getRuntime.exec(Array("gzip", f.getPath))
}

process(new File("."))
```

Once you have a couple of pieces in place, you should be able to write the equivalent in Bash in a couple of lines (but naturally there are many different ways to do this, none of which are wrong. . .)

There are various situations in which Bash will replace special text with some computed values, and this is usually referred to in the docs as `EXPANSION`, e.g parameter expansion, filename expansion etc. Not suprisingly the latter is for finding and inserting files that match a wildcard patterns (this also has the charming moniker `globbing` in shell parlance).

To get an idea of the wildcards you have available, take a look at the Bash man page (http://linux.die.net/-man/1/bash): there is a whole section about different types of Expansion, look specifically "Pathname Expansion".

By default you are limited to a couple of simple wildcards, but when the `extglob` (for extended globbing) setting is enabled you have significantly more options available:

```
$ ls
a.gz  b  c.gz
$ ls !(*.gz)
bash: !: event not found
$ shopt -s extglob
$ ls !(*.gz)
b
```

Note
Why is this behind an option? Backwards compatibility: if you try to use the pattern that you need without setting this option, you will get an error caused by the fact that it already has another meaning.

Another point to note is that the Scala example is descending the directory recursively. One way to achieve this in Bash might be to adopt embrace the Unix philosophy of re-using and combining smaller tools to get the job done. For example, you’ve got `find` (http://linux.die.net/man/1/find) which will do this for you, and you can usefully combine it with `xargs` (http://linux.die.net/man/1/xargs).

Or, again you can delve into the many options of Bash:

```
$ ls *.gz
a.gz  c.gz
$ ls */*.gz
```

```
dir1/a.gz
$ shopt -s globstar
$ ls **/*.gz
a.gz  c.gz  dir1/a.gz  dir1/dir2/c.gz
```

5 Writing scripts

Everything that you write on the Bash command line can be put into a file and executed as a script. The simplest way to run it is just to pass it along as the first argument to a new instance of Bash (the `cat` command will dump the contents of a file to the screen):

```
$ cat hello.sh
echo "hello world"
$ bash hello.sh
hello world
```

Another way to do it is to add the execute permission and add a special line at the top which tells the OS to pass the file to Bash:

```
$ cat hello.sh
#!/bin/bash
echo "hello world"
$ chmod u+x hello.sh
$ ./hello.sh
hello world
$ mv hello.sh hello
$ ./hello
hello world
```

In both of these instances, a new copy of Bash will be run which will house the script while it runs. This means that it won’t be able to access the variables and other things that owned by the starting instance.

In that case, you can either use the `source` command to execute the commands from the file in the same environment exactly as if you typed them at the command line, or you can `export` those resource to child processes.

```
$ ./hello
hello world
$ bash ./hello.sh
Hello
$ cat hello.sh
echo "Hello $NAME"
$ NAME="Superman"
$ bash hello.sh
Hello
```

```
$ source hello.sh
Hello Superman
$ export NAME=Spiderman
$ bash hello.sh
Hello Spiderman
```

6 Exercise Two

Add up the total size of all of the files in a directory.

Search for `Arithmetic Expansion` in the man page to find the way to add numbers together.

You will also find the `stat` command useful for this (although there are pleny of other ways to get this information).

This provides a good example of why shell scripts can often be very fiddly, because things which are easy in traditional programming languages can be difficult in Bash as it requires you to jump through hoops to shift into `programming mode`.

7 Conclusion

We have looked at what Bash is, and just as importantly what it **isn't**. Hopefully you have also gained some signposts that will point the way to getting documention on things you don't understand when reading existing scripts. We have also seem some of the subtlties that can confuse beginners with unexpected behaviour when running scripts.

There are of course a lot more commands and other odds and ends that make up Bash's full toolset, and we will cover these in later sessions.